Politecnico di Milano
Scuola di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica
Academic Year 2010/11

Human-Oriented Event Processing

Human Interactions in Complex Event Processing

Francesco Feltrinelli
francesco.feltrinelli@gmail.com
734505

Supervisor: Gianpaolo Cugola
Assistant supervisor: Alessandro Margara

Contents

List of Figures

List of Tables

Abstract

Human-Oriented Event Processing (HOEP) is introduced as a particular kind of Complex Event Processing (CEP) in which both producers and consumers are human operators: the goal is to widen the CEP domain with scenarios focused on human interaction and collaboration, and vice versa provide people of a given social/business context with the powerful technologies and methodologies of CEP. The significance of HOEP is corroborated by the broad range of applicative scenarios presented. Moreover, the Students-Oriented Event Platform (SOEP), an example of HOEP system addressed to university students, was designed and implemented: students roam around the campus with their mobile phone connected to the system, taking advantage of several services related to their university life. SOEP is composed both by a server part and a client part: the server runs a CEP engine on which event rules are deployed to generate in real-time new events from those published by students; the client is a mobile application run in student's phone which can publish, receive and subscribe to events.

Abstract (Italian)

Lo Human-Oriented Event Processing (HOEP) viene introdotto come un tipo particolare di Complex Event Processing (CEP) in cui sia i produttori che i consumatori sono persone: l'obiettivo è quello di ampliare il dominio del CEP con scenari focalizzati sull'interazione umana e la collaborazione, e viceversa fornire alle persone all'interno di un certo contesto sociale/lavorativo le efficaci tecnologie e metodologie del CEP. La rilevanza dello HOEP è avvalorata dall'ampia gamma di scenari applicativi presentati. È stato inoltre progettato ed implementato un esempio di sistema HOEP rivolto a studenti universitari, lo Students-Oriented Event Platform (SOEP): gli studenti girano per il campus con il loro cellulare connesso al sistema, beneficiando di diversi servizi legati alla loro vita universitaria. SOEP è composto da una parte server ed una parte client: il server si appoggia ad un motore di CEP sul quale sono installate regole di eventi utilizzate per generare in tempo reale nuovi eventi a partire da quelli pubblicati dagli studenti; il client è una applicazione mobile eseguita sul cellulare dello studente che permette di pubblicare, ricevere e sottoscriversi ad eventi.

1 Introduction

Complex Event Processing (CEP) [1] systems emerged in the last years as a particular branch of the general *Information Flow Processing* (IFP) domain [2] to address real-time processing on flows of events, as opposed to traditional computations on data stored on database. Such systems are designed to timely react to the publication of events from several sources in the external world, so that the notified events will be filtered and combined by an event engine to generate new knowledge (in the form of new higher-level - complex - events) according to a set of deployed rules; these generated events are then promptly reported to every interested subscriber. The general idea as presented here has found application in many different domains and have been used in a multitude of scenarios (eg. environmental monitoring, stock analysis, fraud detection) with the common point of processing the information as it flows from the peripheral to the center of the system without requiring it to be persistently stored.

Anyway, although theoretically unrestricted, CEP systems so far have been prevalently used with electro-mechanical components (sensors, actuators), business processes and automated informative systems. The human operator is usually not an active partecipant in the event processing, rather is put by its side as a system manager or data analyst. This thesis tries to go a step father in the direction of human involvement: it introduces *Human-Oriented Event Processing* (HOEP), a particular subtype of CEP which explicitly focuses on human interaction and collaboration, and which features human operators as the main producers/consumers of the event system.

To corroborate the introduction of HOEP, a wide variety of applicative scenarios in HOEP's domain are presented. Moreover a particular HOEP system, the *Students-Oriented Event Platform* (SOEP), was designed and implemented. It focuses on a university campus scenario and aims to provide students with many useful services related to their university life. Students are both the sources and the sinks of the events, and interact with the system through their mobile phone. *Android* (the innovative mobile operating system developed by Google) was chosen as their phone's platform. Section 1.1 gives some more introductory information about SOEP.

1.1 SOEP

The high-level view of `SOEP` is shown in figure 1.1: the student's mobile phone runs `SOEP-Client`, an Android application which lets the user connect to `SOEP-Server` to publish, receive and subscribe to events. On the other side, the server leverages the

event engine `T-Rex` [38], on which `TESLA` [37] event rules are deployed to generate in real-time complex events from simple ones; those are then reported to every subscribing client. `SOEP-Server` and `SOEP-Client` were designed and implemented from scratch, while `T-Rex` is a third-party library which the server merely links to.

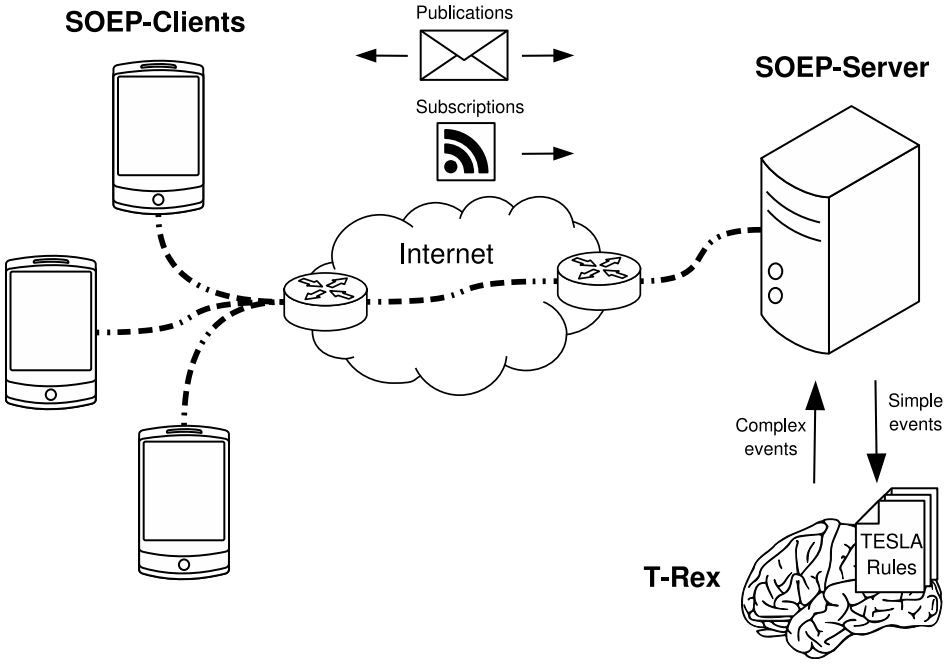

Figure 1.1: SOEP's overview

The introduction of CEP in the university aims to create a portal to many useful services, brought with the contribution of them all. Each service should improve some aspect of the university life and convey a unique user experience: this distinguishes from more traditional event management as in *Wireless Sensor Networks* (WSN) or in the other business domains CEP has been used until now. This care on user experience is needed because the primary actors of the system are students, not sensors: they are not forced to send data or to run the application if they do not want to. Examples of services which could be provided to students through their mobile phone are: real-time information about the crowding of study rooms, facilities to help them in making distributed proposals and to set up agreements, detection of people's gatherings in the campus, management of the queues at the student's office, conducting surveys on university matters.

1.2 Document Structure

The following chapters are devoted to elaborate more on the ideas touched above. Chapter 2 first outlines a brief overview of the Information Flow Processing domain, with a

little reference on historical background and on main models emerged today, starting from Data Stream Processing, continuing to Complex Event Processing, up to the newly introduced notion of Human-Oriented Event Processing. Then it presents the technologies related to IFP which were used in `SOEP`: the `TESLA` rule-definition language and the `T-Rex` event engine. In chapter 3 a broad overview of possible usage scenarios for HOEP is given, both generic and focused on `SOEP`. The latters are supplied with a formal description of the involved events in terms of `TESLA` rules. Chapter 4 gets to the heart of `SOEP-Server`, describing its features and design, with some notes on implementation. Chapter 5 is similarly dedicated to `SOEP-Client`, additionally preceded by a brief introduction on Android. Finally, chapter 6 explores related work and chapter 7 future work. Conclusions are drawn in chapter 8.

2 Information Flow Processing

With traditional *DataBase Management Systems* (DBMSs) the processing of data requires it to be previously stored and indexed; moreover, it is asynchronous with regards to its arrival, that is, data is first stored and on a later time requested by database's clients. These peculiarities made DBMSs unsuitable for scenarios where the timely processing of a continuous flow of information from the peripheral to the center of the system is required, especially when many distributed sources and sinks are available, and data arrival time is unpredictable; these systems could be collectively referred to with the umbrella term *Information Flow Processing* (IFP) [2]. The focus is on real-time processing of a flow of information according to a set of deployed *rules*, which describe how to create new knowledge from detected pieces of information. In such situations (think for example about a fire detection system, which uses sensors placed around a given geographical area to monitor critical environmental conditions), there is usually neither time nor need to store everything persistently on disk, because irrelevant information can be immediately discarded, while relevant one can be retained in memory only until it is used to produce new knowledge, and freed after that.

In section 2.1 the main models in IFP domain will be briefly described. Section 2.2 focus on the subset of IFP technologies used in `SOEP`.

2.1 IFP Models

Several communities - including those specialized in distributed information systems, business process automation, control systems, network monitoring and sensor networks - brought contributions to the IFP research branch, exploiting the new methodologies and technologies to tackle the most disparate scenarios, eg. in environmental monitoring [21], stock ticker monitoring [4], credit card fraud detection [8], intrusion detection [19], RFID-based inventory management [12] and manufacturing control systems [13][14]. Pushed by this multitude of contributions from different points of view, though with a common goal, several systems were created which differ in architectures, data models, rule languages, and processing mechanisms [2]. Nowadays two main models emerged and can be denoted as: *Data Stream Processing* [30] and *Complex Event Processing* [1]. This thesis introduces another one, the *Human-Oriented Event Processing*, as a particular kind of CEP.

2.1.1 Data Stream Processing

In the *Data Stream Processing* (DSP) model, information coming from sources is seen as a stream of data to be processed in order to produce a new stream of data as output. This kind of systems were designed as extensions to traditional DBMSs to manage online, unordered and unbounded flows of generic data.

The first result in this direction was given by *Active Database Systems* [31], which were introduced to allow *actions* to be automatically executed when given *conditions* arise. As traditional DBMSs are completely passive, because they only react to an explicit data request made by the application, it is not possible to ask them to asynchronously perform some processing when a given condition verifies: the application itself must implement its own logic to obtain that. Active database systems were developed to overcome this limitation: they can be seen as an extension of classical DBMSs where the reactive behaviour can be moved from the application to the database. The sources of the events which could trigger an action may be internal operators, like a tuple insertion or update, or in some cases also external events, like those raised by clocks or sensors. Similarly, the action taken could be only internal as an update on the database, or sometimes external like the notification of the application.

The limit of active database systems in the context of IFP is their need for a persistent storage where all the relevant data is kept, whose update is assumed to be relatively infrequent. As a consequence, their performance sinks under the weight of a high number of queries or high rate of events' arrival. *Data Stream Management Systems* (DSMSs) [30] went farther to solve these issues and to allow query processing in the presence of continuous unbounded data streams. Differently from a DBMS, which requires data to be stored persistently with the assumption of infrequent updates, a DSMS deals with volatile unpredictable streams of data. Another difference is on the semantics of queries: while in a DBMS queries are *one-time*, that is, evaluated once over a snapshot of the data set, with the answer returned synchronously to the caller, a DSMS executes *standing* queries, which run continuously from their deployal to their removal, asynchronously providing results as soon as new data comes. [2]

Despite these differences, DSMSs have much in common with DBMSs, especially in the way they process data through common SQL operators like selections, aggregates, joins, and all the operators defined by relational algebra.

2.1.2 Complex Event Processing

While in DSP the information flowing in the system is treated as plain data, which the application is in charge of assigning a semantics to, the *Complex Event Processing* (CEP) model [1][2] characterizes the information items as *events* coming from the external world, each associated with a particular semantics. Another difference between DSMS and CEP is that the former usually manages *homogeneous* information flows, as uniformly-typed data streams which fill transient unbounded database tables, while the

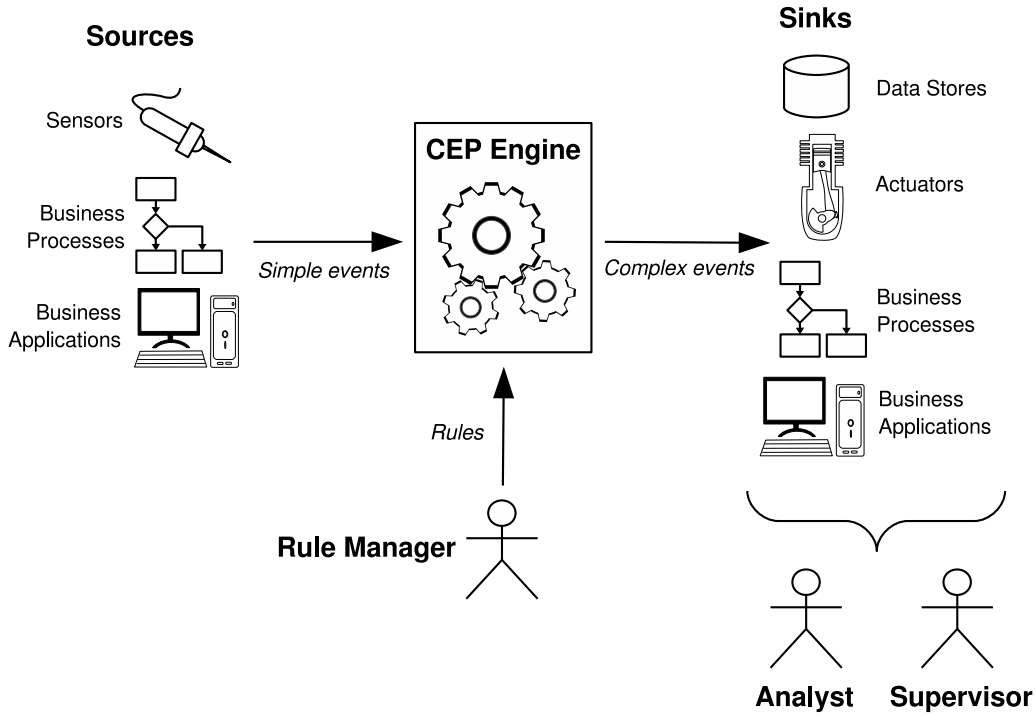

Figure 2.1: CEP system's overview

latter allows the processing of *heterogeneous* events coming from a multitude of different sources.

In CEP an event *engine* takes as input this flow of low-level (also *simple*, or *primitive*) events coming from *sources*, combines them according to a set of pre-installed *rules*, and creates new higher-level (or *complex*) events when particular *patterns* of simple ones are matched; the complex events are then returned as output to every interested *sink* (see figure 2.1). Note that the CEP engine may have a simple centralized architecture, as well as adopt a distributed architecture with a set of *Event Processing Agents* (EPA) connected in an *overlay network*.

The CEP model can be seen as an evolution of the *publish-subscribe* model: while the latter considers singularly each event to be delivered, in the former elaborate patterns of multiple related events are defined. In addition, with CEP *hierarchies* of events can be built, because a complex event can be in turn used to define another higher-level event, and so on[1]. This pyramid of events is an effective way to organize knowledge, and ease the design of an event system with the creation of an *abstraction* on the core concepts of the application's domain.

Although in theory producers (consumers) in CEP can be anything that produces (con-

[1]Actually not all the rule-definition languages are expressive enough to allow this chaining of events; `TESLA`, the language adopted by `SOEP`, does it.

sumes) an event, in practice traditional CEP systems (as shown in figure 2.1) have been used so far with electro-mechanical components (sensor and actuators of various kinds), automated informative systems, business applications and business processes. As regards the latters, the combination of CEP and *Business Process Management* (BPM) is sometimes called *Event-Driven BPM* (ED-BPM) [32]: the idea is to use event processing to detect situations, and the BPM part to react by triggering a new BPM workflow instance, or by updating an existing one. In all these cases the human operator is not an active partecipant in the event processing, rather is put by its side to carry out system management (eg. *Rule Manager* and *Supervisor*) or data analysis (*Analyst*).

Rule managers deploy event rules, which can be defined in any ad-hoc language (eg. `TESLA`), and can impose various types of *constraints* on the pattern of events to be detected. Depending on the expressiveness of the language, the definable constraints may regard the temporal relationships among a sequence of events, and filters on their content. Content-based constraints may involve single events (for example, the value of an event's attribute could be imposed inside a certain range), or use an *aggregate* on set of events, that is, a function applied on them which generates a constrainable value.

2.1.3 Human-Oriented Event Processing

Human-Oriented Event Processing (HOEP) is a particular type of CEP in which both sources and sinks are human operators, which interact with the system through fixed terminals or more often mobile devices. HOEP focuses on collaboration between people, aimed to efficiently pursue a common goal. This interaction could be set in working life, that is, while a business process is being carried out by human workers, or it may involve aspects of private life, as sociality, fun, hobbies. The term *human-oriented process* is used to refer to an automated process in which human role - as input and/or output - is essential.

HOEP's domain has some common elements with *Human Interaction Management* (HIM) [33], but it largely differs from it. HIM draws a framework to integrate in the context of BPM all those processes performed by people which require innovation and creativity (eg. research, product design, marketing [34]) and as such cannot be managed by traditional BPM techniques. On the other hand, HOEP focuses on systems whose core logic is automatable - in a similar way to BPM - but whose front-end and back-end are made by people. Of course also in HOEP human operators interact with the system through devices and applications, but the flow of information they generate is immediately referable to them.

2.1.3.1 Characteristics

There are some characteristics of HOEP which distinguish it from traditional CEP. The first one, as it was said, is its being carried out by both human producers and human

consumers. An immediate consequence is that the *quality* of the reported events is subject to high variance: people can easily make *mistakes*, or even *lie* (eg. for competition, sabotage, etc.). Note that even in traditional CEP the management of uncertain, inexact events coming from unreliable sources or over an unreliable channel, may become an issue [1, sec. 11.2]; anyway, human unpredictability makes the demanding techniques proposed so far (probability-based methods such as Monte Carlo algorithms or Bayesian networks [35], evidential reasoning as in Dempster-Shafer theory [36], fuzzy logic, neural networks) even harder to be applied in HOEP's context. Moreover, people may be *unwilling* to collaborate: after all, people are not automata. Their *benefits* from partecipating in the event system should balance the drawbacks (mostly the time spent to learn and to use it, sometimes also the cost of its development, implementation and maintenance): the service should be useful and attractive. It is important to create a high degree of *involvement*.

The second major characteristic of HOEP is that involved people are usually part of the same social/work *group*, or related groups. They are all collectively engaged in reaching a common goal, or at least in doing their part well to eventually benefit from it. The quality of the resulting service depends on the quality of individual *contributions*: from this it becomes clear the importance of the *collaborative* aspect.

Another feature that poses some challenges over traditional CEP is the usage of *mobile phones*. As almost everyone nowadays owns a smartphone, it seems to be the ideal vehicle of interaction with the event systems, but there are some caveats. First of all, phone's *connectivity* is highly variable: with varying degrees depending on the scenarios, connection with the system cannot be supposed constantly available, and there should be a mechanism to allow offline work and periodic synchronization. Another drawback of mobile phones over other kinds of infrastructural equipment is the limited *battery life*, which makes it necessary to reduce energy consumptions as much as possible[2]: both the hardware and the software on the device should be optimized to avoid wastes of energy. Moreover, the mobile application used to interact with the HOEP system will be installed on a *non-exclusive* platform, as the phone will be shared with other applications. Thus, it is important to design it to integrate seamlessly and in a non-invasive manner with the mobile operating system; this also includes constraints on computational resources' usage, as the application should not stall the others. Finally, to reach the vastest amount of users, the application should be *cross-platform* or different versions for the main mobile platforms should be released. Even with a single platform, differences in hardware equipments should be taken into account: a phone could have a hardware configuration as minimal as a keyboard, screen, microphone, cellular network module[3] and camera, going on with the addition of a touchscreen, Bluetooth, Wi-Fi and GPS, up to many sophisticated sensors like accelerometer, gravity, magnetic field, gyroscope, ambient light, proximity; much more seldom are temperature and humidity sensors.

[2]The same need arises in the WSN domain

[3]Used both for voice and data link; technologies for cellular connectivity span from 2G (GSM, GPRS), 3G (EDGE, UMTS, CDMA2000, Mobile WiMAX), and the upcoming 4G technologies.

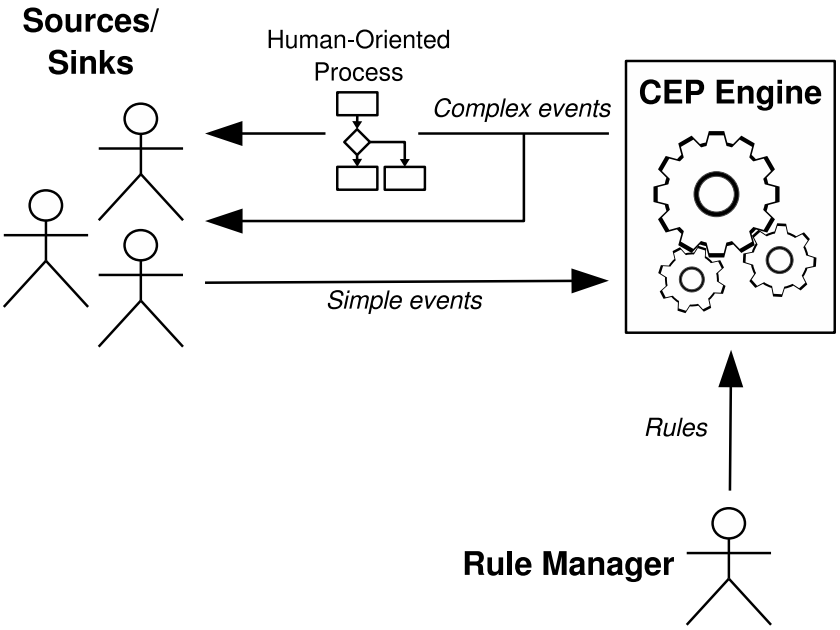

Figure 2.2: HOEP system's overview

2.1.3.2 Architecture

In figure 2.2 you can see the general architecture of a HOEP system. Compare it with traditional CEP systems (figure 2.1): here producers and consumers are people. They are not limited to side roles, as system management or data analysis, rather they are active actors in the event processing. The complex events generated by the CEP engine can be sent to human operators directly, or they can trigger the creation/update of automated human-oriented processes which will in turn produce new information relevant to them.

2.2 IFP Technologies in SOEP

`SOEP` suits better to the HOEP/CEP model rather than the DSP model: the capability of inferring new knowledge from the information received from sources is at the basis of the services it provides to students. By using the students as both sources and sinks of the system, those services can be tailored on their needs and react cleverly on their input. It is not a matter of processing a stream of data as in DSMSs: here sources are numerous and provide a high volume of information items whose order and time relationships may often be relevant. Moreover, the ability to define expressive hierarchies of events is useful to filter and combine those items at different logical levels to produce meaningful results.

The following sections focus on two CEP technologies which were also adopted in `SOEP`: section 2.2.1 introduces `TESLA`, a rule definition language, while section 2.2.2 describes `T-Rex`, the CEP engine which `SOEP-Server` is built on.

2.2.1 TESLA

The Trio-based Event Specification LAnguage (`TESLA`) is a complex event specification language designed by G. Cugola and A. Margara at Politecnico di Milano [37]. `TESLA` is highly expressive despite its simple syntax and has formal semantics, given in terms of the first order, metric temporal logic *Trio*. It introduces a limited number of operators, through which it provides content and temporal constraints, parameterization, negations, sequences, aggregates, timers, and customizable policies for event selection and consumption.

A `TESLA` rule defines a complex event from its constituting simple events. Each event has an associated *type*, which defines the number, order, names, and types of its *attributes*, and a timestamp. The general structure of a rule is the following:

```
define      ComplexEvent(Attribute1: Type1, ... , AttributeN: TypeN)
from        Pattern
where       Attribute1= f1, ... , AttributeN= fN
consuming   e1, ... , eN
```

where the `define` clause declares a new complex event specifying its type, and the `from` clause describes the pattern of simpler events that lead to the complex one. In the `where` clause the actual values for the attributes of the new event are given, using a set of functions `f1`, ... , `fN` which may depend on the arguments defined in *Pattern*. Finally, the optional `consuming` clause lists the events that will not be used again next time the rule will be fired.

Given the easy syntax of the language, it is quite intuitive to understand its semantics: for the purpose of this document some concrete examples of rules - presented in the following - will be enough; you can refer to [37] for a more formal and complete discussion on `TESLA`. The example presented here mimics an environmental monitoring application, where sensors periodically notify their position, temperature and smoke presence. From the readings of the sensors the system uses this rule to deduce when a fire alarm should be launched:

```
define      Fire(area: string, temperature: double)
from        Smoke(area=$a) and each Temp(area=$a and value>45) within 5 min. from Smoke
where       area=Smoke.area and temperature=Temp.value
```

This rule states that a fire event should be created whenever a smoke notification is received in a certain area and for every temperature reading with value greater than 45 degrees read at most 5 minutes earlier in the same area. The fire event is supplied with the indication of the interested area and the temperature detected. Note how the occurrence of a complex event (*Fire*) is bound to the occurrence of a simple event (*Smoke*), which acts as an anchor point and implicitly determines the time at which the new event is created. This anchor event (also denoted as the *terminator* of the sequence) is coupled with other events (*Temp*) through sequence operators (`each-within`) which express the temporal relationships among them. Moreover, event selection is restricted by the *parameter $a*, which imposes the area of *Smoke* to be the same as that of *Temp*.

The `each-within` operator defines a *multiple selection policy*, because it creates a new complex event for every event of the specified type in the given time *window*; `first-within` and `last-within` operators are also available to define *single* selection policies (the former selects only the first event, the latter only the last one). It is also possible to use *aggregate* operators, which are functions computing a value from a set of events, useful either to filter out events that do not match certain conditions, or to give values to the attributes of a newly created complex event. It should also be mentioned that it is possible to schedule the periodic evaluation of rules through a special event called *Timer* used in the `from` clause; this way the creation of complex events can be driven by a regular timer instead of by the reception of a rule's terminator event.

A final note on *subscriptions*. Subscriptions in `TESLA` are as simple as in traditional publish-subscribe languages and include the type of the event together with a filter over the content of its attributes, for example:

```
Subscribe(Fire, area= "Building 20" and temperature> 50)
```

Subscriptions may refer either to simple events (those directly observed by sources), or to complex ones (those derived through `TESLA` rules).

2.2.2 T-Rex

`T-Rex` is a general-purpose CEP engine developed by A. Margara and G. Cugola at Politecnico di Milano [38]. It supports `TESLA` rule-definition language and uses efficient processing mechanisms. It is *free* software - released with LGPLv3 licence - implemented in C++ but with Java client adapters also available. `SOEP-Server` uses `T-Rex` as its CEP engine.

A client interacts with `T-Rex` by installing processing rules, publishing events and listening for the creation of new events. `TESLA` rules are compiled into efficient in-memory data structures and dynamically processed as simple events are submitted. A listener is registered on the engine to be notified when new complex events are created, as a consequence of the processing of simple events according to installed rules.

For a more detailed overview on `T-Rex` you can refer to [38] or go to the project's web site (`http://home.dei.polimi.it/margara`). Section 2.2.2.1 gives a brief description of the processing algorithms used.

2.2.2.1 Processing Algorithms

As it was explained in section 2.2.1, a `TESLA` rule describes a sequence of primitive events which must be detected to generate the complex event it declares, and imposes constraints on their content and their occurrence time. Moreover, the rule binds the occurrence time of the complex event to the detection of the sequence terminator. The goal of `T-Rex`'s processing algorithm is to analyze the history of received primitive events, looking for the sequences which a composite event should be generated from.

Two opposite approaches can be followed as regards the detection of relevant sequences. On one hand, the engine could temporarily buffer incoming primitive events, postponing all the processing to detect sequences until a terminator event is submitted to the engine. On the other hand, events can be incrementally processed as they arrive, and only the results of the intermediate computation temporarily stored.

The algorithm corresponding to the first approach has been called *Column-based Delayed Processing* (CDP), since it organizes the history of received events into sorted columns and delays their processing until a terminator comes. Each rule keeps a column for each primitive event appearing in the sequence it defines, and store events on their corresponding column as they are received. When a terminator is received, a backward iteration which analyzes columns' content starts from last column (the one associated with the terminator) to first one (associated with the first event of the sequence), gradually discarding unsuitable events and composing partial sequences of events which are good candidates to form a relevant terminal sequence. When the iteration reaches the first column, the remaining sequences are the valid ones and a complex event is created for each of them.

The algorithm that follows the second approach was denoted as *Automata-based Incremental Processing* (AIP), because it processes events incrementally as they arrive and stores partial results as automata. Each rule is translated into an *automaton model*, which is a linear deterministic finite-state machine where each state is mapped to an event in the sequence defined by the rule. A new instance of the automaton model, with current state one step forward in the sequence, is created for every received event which satisfies the type, content and time constraints necessary to trigger the transition. When an automaton instance reaches last state (the one mapped with the terminator event) a valid sequence has been recognized and the corresponding complex event is generated.

`T-Rex` has been implemented with both AIP and CDP algorithms; it makes extensive use of multi-core processing and exploits several other optimizations to improve overall efficiency. It is also available an implementation with CDP optimized for *CUDA*, a widespread architecture for general purpose programming on GPUs, developed by Nvidia. Note that all of these versions are centralized; a distributed version, which decomposes `TESLA` rules to process parts of them on different nodes of an overlay network, is also in progress.

`SOEP-Server` was tested with both the CPU-based AIP and CDP versions, and currently adopts the CDP one. Refer to [39] for a more detailed description of the processing algorithms used in `T-Rex`.

3 Scenarios

In this chapter lots of scenarios for a HOEP system (see section 2.1.3) are presented. Scenarios in section 3.1 come from different and unrelated contexts, while section 3.2 focuses on scenarios specifically targeted to `SOEP` (see section 1.1 for an introduction on `SOEP`, or chapters 4 and 5 for a detailed description of the server and client part, respectively).

3.1 Generic Scenarios

This section contains a wide variety of possible scenarios for a HOEP system, coming from many unrelated contexts. The same generic structure depicted in figure 3.1 will be used to describe each of them: it includes the *input events* sent by sources to the event engine, the *processing* done by the engine in reaction to those events, which eventually will end up with the generation of new *output events* delivered to sinks. Sometimes there are more than one work team or social group involved, and there may be ambiguity on which of them the reported events refer to: in these cases the used syntax is ‹*source*› → ‹*event*› to specify the source of the event, and ‹*sink*› ← ‹*event*› to specify the sink of the event. All the scenarios are assigned one or more high-level *categories* to better organize them.

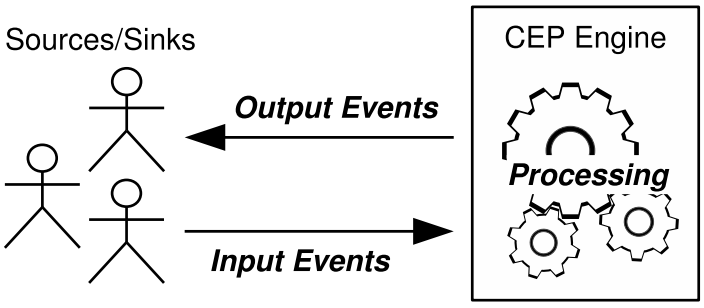

Figure 3.1: Scenario's framework

3.1.1 Radio-Taxi

Categories: [*Business Process Management*]

Input Events	Processing	Output Events
customer → ride request	choose available taxi	customer ← estimated waiting time
taxi → position	estimate service time	taxi ← ride assignment
taxi → ride started	choose waiting area	taxi ← waiting area assignment
taxi → ride finished		

Table 3.1: Events in *Radio-Taxi* scenario

Management of a private network of radio taxis with a HOEP system (table 3.1) [1, sec. 1.5]. Its goals are to effectively assign ride requests to taxis spread all over the city, and to supervise the deployment of the units on the various reserved waiting areas, to guarantee best coverage. The taxi is connected to the central system through radio equipment, which is also provided with GPS to periodically signal its position. A customer requests a ride either with a phone call or by pushing a button on a dedicated pedestal-style box. The request is routed to the system, which chooses a taxi among those available and close to the pick up point, assigns it the ride, and informs the customer about the estimated service time. The taxi notifies the start of the ride when it picks up the passenger, and its conclusion when it gets to the destination. After the ride is finished the system sends the taxi to one of the waiting areas to wait for next ride.

3.1.2 Door-to-door Salesman

Categories: [*Business Process Management*]

Input Events	Processing	Output Events
salesman → selling report ▷ customer ▷ address ▷ proposed products ▷ proposed prices ▷ selling outcome	update areas to scour	salesman ← area assignment
	update products' priority	salesman ← products' priorities
	update products' price	salesman ← products' prices
	update salesmen's score	salesman ← score report
warehouse operator → stocks update		
marketing consultant → market prices		

Table 3.2: Events in *Door-to-door Salesman* scenario

Management of a network of door-to-door salesmen (table 3.2). It should help to organize salesmen's work by effectively spreading them around the city and notifying them about the priorities and prices of the products they are trying to sell. Salesmen are connected with the system through their mobile device, and send a report after every sale's attempt,

which includes the customer met, the products suggested along with the proposed prices. The assignment of the areas to be scoured takes into account the current deployment of salesmen, their timetables, the sales' volume reported in each area and how much it has already been combed in the recent past. Products' priorities and prices are updated according to the sales done by the salesmen, the current warehouse's status, and real-time evaluations of market prices. As salesmen are rewarded on the basis of their sales, statistics are made from their reports which lead to the update of a dedicated score system.

3.1.3 Road Traffic

Categories: [*Business Process Management*, *Public Service*, *Time Management*]

Road traffic identification and control (table 3.3). Cars are provided with a satellite navigator which takes into account traffic conditions and reroutes the driver when an issue along its path has been reported. The system detects roads' congestion starting from the position and speed of the vehicles (GPS) and their proximity with each other (proximity sensor). While GPS and proximity readings are automatically sent to the system by the car's electronic equipment, drivers can also manually report road issues through the touchscreen of their navigator. The system can also combine information about the current traffic situation with historical data to make short-term traffic forecasts and alert the user about possible delays.

Input Events	**Processing**	**Output Events**
car position	compute real-time traffic	traffic update
car speed	compute traffic forecast	alternative route
car proximity		
road issue report ▷ car accident ▷ disabled vehicle ▷ road damage ▷ broken traffic signal		

Table 3.3: Events in *Road Traffic* scenario

3.1.4 Foot Traffic

Categories: [*Business Process Management*, *Time Management*]

Suppose a big chain store (eg. a supermarket chain) wants to provide their customers with a way to monitor the real-time traffic on their stores and mean service times, in order to choose the least busy one, give up and try another day, or at least consciously

face the queue and organize other appointments (table 3.4) [40]. The crowd on each store, as well as the mean waiting time, is inferred from the readings of people counters placed above the entrance combined with the transactions from the electronic cash registers. Customers may check the current situation through an interactive map on the company's website, or install a dedicated mobile app on their phone.

Input Events	Processing	Output Events
cash register → transaction	compute real-time waiting time	waiting time update
people counter → client in		
people counter → client out	compute waiting time forecast	

Table 3.4: Events in *Foot Traffic* scenario

3.1.5 Tournament Organization

Categories: [*Collective Scheduling, Personal Information Management, Time Management*]

A system to help the organization of sports tournaments, eg. a tennis tournament (table 3.5). Meant to support amateur competitions, which do not have full-time dedicated organizers, the system relies on players's input from their mobile phone to carry on the automatic scheduling of the matches and update the tournament's scoreboard. Players notify changes on their availability (in terms of the days they can play and their preferred time slot), and announce the result of a match just played. After some processing, the system schedules the next matches and alerts the involved players, as well as broadcasts the current players' scores. When a new match is notified, the players' mobile app automatically adds the event to calendar and the opponents' contacts on the address book.

Input Events	Processing	Output Events
player's availability	compute matches' schedule	next match
match result	update scoresboard	scores

Table 3.5: Events in *Tournament Organization* scenario

3.1.6 Geocaching

Categories: [*Social, Fun*]

Geocaching is a modern form of treasure hunting: participants are defied to find small objects (called *geo-caches*) hidden somewhere, usually with the help of their GPS phone

or dedicated satellite equipment. There are many websites[1] which collect and organize the caches published by users, ordered by difficulty or geographical area, and let the community enrich them with social content, multimedia clips, geotags, and so on[2]. In the most basic modality the participant chooses one of the challenges proposed in the preferred area, takes note of the cache's coordinates (sometimes encoded in a QR code[3]) and go hunting near that location to find it. Once he has found it, he usually writes the date and his sign over a small logbook contained inside the cache, and hides it again; he can also notify his success on the website and add comments about the hunt. Larger caches can also contain items for trading, usually toys or trinkets of little value. More challenging hunts involve the interpretation of clues and riddles to find the cache, sometimes even with intermediary chained caches which eventually lead to the final one.

As presented above, the social part of the game is concentrated more in the sharing of contents on the website, rather than in the act of hunting itself, which is configured as an individual challenge (though participants may join in small groups to look for the same cache together). The following scenario proposes a modality for geotagging which put more social emphasis on the hunt, similarly to the traditional treasure hunting (table 3.6).

Input Events	**Processing**	**Output Events**
position	verify cache validity	next cache ▷ exact coordinates ▷ approximate position with clues/riddles
cache discovered	choose next cache	
	update scoresboard	scores

Table 3.6: Events in *Geocaching* scenario

Many treasures are scattered around the play area (being it a small garden or an entire city) before the game starts. The hunt is done in teams, which challenge each other to find the most number of them (possibly treasures may be given different values). Each team has a GPS mobile device connected with the system, which periodically sends its position. When a team discovers the current cache, it sends the contained unlock code to the system, which verifies it and - if correct - chooses the next one and communicate it. The next cache is dynamically chosen according to the current teams' position, in order not to favour any of them in terms of geographical proximity. The scoreboard is updated after every discovery and notified to teams.

[1] The most famous one is `www.geocaching.com`

[2] Including, of course, buying caches and other similar gadgets

[3] A particular kind of matrix barcode which can be scanned and decoded by a dedicated reader or the mobile phone's camera

3.1.7 Laser Tag

Categories: [*Social, Fun*]

Laser tag is a team activity where players hold infrared-emitting guns and score points by tagging other players. Infrared-sensitive targets are worn by each player and sound when being tagged, so that the hit player has to go out of the arena. The match may be played either indoor or outdoor with different modes, among which:

- ▷ *team deathmatch*: each team has its own hit-count, which is incremented each time one of the team mates tags an opponent. The team with the highest hit-count at the end of a fixed time interval wins;
- ▷ *capture the flag*: each team tries to steal the opponents' flag from their base and take it back to its own base in order to score a point or win the match;
- ▷ *protect the VIP*: there is a special player called the VIP, who only wears the infrared target but has no gun; the team with the VIP must conceal and protect him for a set length of time while the opposing team tries to eliminate him;
- ▷ *conquer the base*: a team must defend its base while simultaneously attacking the opponent's one; points are gained for possessing the target base for certain lengths of time.

Input Events	**Processing**	**Output Events**
position	update scoresboard and stats	scores and stats
gunshot	check game end	game start/end
player/VIP hit	schedule next game	new game mode / new target
flag captured		

Table 3.7: Events in *Laser Tag* scenario

A laser tag session could be supported by a HOEP system (table 3.7). The player's infrared equipment -as well as other mobile or infrastructural elements such as the flag or fields' stations - would be radio-connected with the system to send all the relevant events (RFIDs could also be used in indoor environments), in addition a GPS receiver would periodically transmit the player's position. Scores would be updated according to the gunshots fired and the targets hit, or the position mantained inside constrained areas (eg. the opposing team's base), depending on the game mode. Players would be notified of the start/end of matches and about dynamic changes of the game mode or the choice of a new team's target.

3.1.8 Bingo

Categories: [*Social, Fun*]

Bingo is a famous game of chance in which randomly drawn numbers are matched by players against numbers' matrices (called *cards*) which players buy before the game

starts. A jackpot is made available as a percentage of the total money got from cards' sale, and money prizes are assigned to specific patterns achieved on the cards from the drawn numbers (eg. five numbers in a row, or all the card's numbers matched, called *bingo*). Usually cards are in printed format.

The game could also be electronically managed by a HOEP system (table 3.8). Players are provided with a tablet or use a fixed terminal, connected to the event system. New virtual cards can be bought at the beginning of each match; cash money is paid only at the exit of the room, or the credit card is used. As much of the fun in the game consists in detecting and announcing a win to the bank, this aspect should not be automated: players have to take care of their cards and manually spot a matching pattern; when they found one, they notify it by pressing on their touchscreen. The system then checks the validity of the announced pattern and possibly confirms the win.

Input Events	**Processing**	**Output Events**
pattern matched ▷ five in a row ▷ bingo ▷ other patterns	draw next number	drawn number
	verify pattern	win confirmation
	update jackpot	
buy new card		

Table 3.8: Events in *Bingo* scenario

3.1.9 Concert Setlist

Categories: [*Social, Collective Scheduling*]

While nowadays concerts have a fixed setlist pre-determined by the band, it would be more involving for the audience to be able to vote in real-time for their favourite songs to be played, much like a televoting system (table 3.9).

Input Events	**Processing**	**Output Events**
song's vote	choose next song ▷ compute most voted song ▷ leave out already/just played songs ▷ apply band's contraints	next song
		songs chart

Table 3.9: Events in *Concert Setlist* scenario

Before the beginning of the show, the band declares the list of the songs it is willing to play, along with any possible constraints on the voting (eg. fixed tracks in the setlist,

non-consecutive tracks, etc.). The vote is done dynamically, song by song: while the band is playing a track, the spectators vote for the next one to be chosen, through a dedicated mobile app or simply with an SMS. The real-time results of the televoting, with the most wanted songs' chart, is displayed on big screens next to the stage.

3.1.10 Personal Sensing

Categories: [*Social, Human-Computer Interaction*, *Context-aware Computing*]

Input Events	**Processing**	**Output Events**
smartphone → sensor reading ▷ accelerometer ▷ position (GPS-based or network-based) ▷ ambient light ▷ microphone ▷ camera ▷ proximity ▷ WiFi scan	infer user's activity ▷ standing / sitting ▷ walking / running / driving ▷ listening to music / watching movie ▷ talking / phone calling / chatting / writing mail	profile update avatar update
smartphone → OS event ▷ audio/video playback ▷ phone call / chat / mail	infer user's environment ▷ position (raw coordinates or geolocated) ▷ indoor / outdoor ▷ sunny / cloudy / rainy ▷ at home ▷ at work ▷ at a concert ▷ crowded place ▷ close to friends	
PAN device → sensor reading ▷ humidity ▷ temperature ▷ galvanic skin response		

Table 3.10: Events in *Personal Sensing* scenario

A *personal sensing system* [41][42][43] used to infer the current user's status (his activity, geo-climatic and social context) and properly update in real-time his profile on a SNS[4] (eg. *Facebook*) or his avatar in a virtual world (eg. *Second Life*). The system (table 3.10) combines a CEP engine with sophisticated activity-inference techniques to deduce the user's activity and environment basing on mobile phone's sensors readings, possibly combined with other PAN[5] devices (eg. a Bluetooth fitness monitor supplied with GSR sensor[6]) or infrastructural nodes (eg. RFID readers spread on key city locations). User's

[4] *Social Network Service*

[5] *Personal Area Network*

[6] *Galvanic Skin Response* (GSR), also known as *skin conductance*, is a method of measuring the electrical conductance of the skin, which varies with its moisture level. As the sweat glands are controlled by the sympathetic nervous system, GSR is used as an indication of psychological or physiological arousal

phone could also interact with other mobile devices in the neighbourhood, for example to automatically detect the presence of friends, or to compensate a limitation on its sensors (either permanent, as the lack of a GPS receiver, or temporary, as the inability to receive GPS signal due to indoor position) borrowing readings from other phones. Personal information (such as address book contacts, calendar events, and so on) could also be sent and stored to detect known/favourite people/places and make statistics on user's habits in order to improve its status' inference; of course this would require full-customizable privacy settings in the mobile app.

3.2 SOEP's Scenarios

In the following sections some usage scenarios for `SOEP` are described. Each scenario starts with a preamble which contextualizes the example given and justify the need for a `HOEP` system, followed by some possible `TESLA` rules (section 2.2.1) that would implement the described service. The scenarios refer to *Politecnico di Milano* as an example of university which could benefit from `SOEP`.

An effort has been made to present in this document scenarios that could be implemented only with the student's mobile phone connected to the server: no additional hardware is required. The purpose is to show the potentials of a `HOEP` system and the improvements it could make even without expensive infrastructural changes. An attempt has been done to find out applications useful to students, who are the primary users of the system. In the future many other innovative services could be conceived by integrating more deeply with university's infrastructures.

3.2.1 Crowding in study rooms

A big nuisance for students in Politecnico di Milano is to find an available room to study in. There are various study rooms spread all over the campus, but available seats are barely enough to let everyone in. Students must roam around the campus from one room to the other, till they eventually find a place to stay.

In this context, an automatic service that let users know about real-time crowding in study rooms would be useful. This service would collect positions of the students and aggregate data to compute crowdings for each room. This should be done automatically, including the notification of student's position in a certain room: it is not realistic to suppose that a student would do it manually any time he stops in a room. Moreover, we would like to do it without additional hardware (eg, RFIDs on each study room, or turnstyles accepting students' cards) other than the Android mobile devices owned by the students and the server.

Android mobile devices have two ways to know their position: the best way is the GPS that is built-in the device, which has an accuracy up to a few meters under good

conditions; a second way is to get their location from the cell towers of the mobile network, but it is a very coarse estimate with an accuracy that can be a few hundred meters. It is clear that the second way is too coarse for our purpose, but also GPS is useless because we are talking about indoor environments.

Another custom way of determining student's position could be used. In Politecnico di Milano any study room is provided with WiFi access, and for many of them the Access Point (AP) is dedicated. Thus, the BSSID (that is, the MAC address) of the AP could be exploited: student's mobile periodically scans for WiFi networks and sends their BSSID to the server; the former knows which AP covers which room, and can realize if the student is in one of those rooms. Note that student's devices only scan the air and get information about each wireless network in range, but do not need to connect to them.

The event sent by Android devices would be something like:

```
WifiDetection(BSSID=[address of the AP of a network in range])
```

where *BSSID* is the unique identifier of the AP of a network just detected. WifiDetection events are sent automatically and periodically (eg. every 5 minutes or more) by the mobile phones, so the same WifiDetection is repeated if the student keeps staying in the room, while it is just not sent anymore when the user leaves the room.

The system is set to periodically count the number of people in each room:

```
define    RoomXPeople(PeopleNum)
from      Timer(M%5==0)
where     PeopleNum=Count(WifiDetection(BSSID==[bssid of room X])) within 5min from Timer
```

with a rule for each room (X is the room identifier). *Count* is an aggregate operator that returns the number of occurred events of specified type. Actually, with this approach rules are not parameterized on room identifier (and each rule is for one room only) because of the checks on the BSSID, which cannot be done all in a single rule. Automatic generation of the rules should overcome this nuisance.

In case a more qualitative indicator would be needed (rather than the approximate number of people in the room), other higher-level rules could be exploited:

```
define    RoomXCrowding(Crowding)
from      RoomXPeople(PeopleNum≤5)
where     Crowding="LOW"
```

and

```
define    RoomXCrowding(Crowding)
from      RoomXPeople(PeopleNum>5 and PeopleNum<15)
where     Crowding="MEDIUM"
```

and

```
define    RoomXCrowding(Crowding)
from      RoomXPeople(PeopleNum≥15)
where     Crowding="HIGH"
```

that use textual indicators instead of numbers. Again, there is a triple of rules for each room, because each room has a different capacity and therefore its specific rule has knowledge of what are the criteria to define its crowding.

To simplify user's subscription, a rule that aggregates notifications on rooms is useful:

```
define    RoomsCrowding(Room1Crowding, ... , RoomNCrowding)
from      Timer(M%5==0) and
          last Room1Crowding() as R1 within 5min from Timer and
          ...
          last RoomNCrowding() as RN within 5min from Timer
where     Room1Crowding=R1.Crowding,
          ...
          RoomNCrowding=RN.Crowding
```

so that the user can subscribe with:

```
Subscribe(RoomsCrowding)
```

and periodically get notifications about the crowding in every room.

Of course, this way of getting users' positions through the BSSID of WiFi networks is intrinsically approximate, and some mechanisms should be adopted to prevent easy errors like, for example, students in range with the network but not stopped in the room (thus not occupying available seats), maybe just walking near it or stopped in the adjacent room which is also in range with the network, and so on.

3.2.2 Proposals and agreements

A `HOEP` system could also be used as an instrument to make proposals and set up agreements. Imagine a student during a typical university day: many hours of lessons distributed during the day, and here and there some break time. Often breaks are used just to relax, but sometimes he would like to fill one of those with something funny. For example, he could want to arrange a small football match with other students: the event he sends to the system is then actually a proposal of the type:

```
FootballMatchProposal(Name=[student's name], When=[proposed time])
```

that is a simplified example in which just the student's name and the proposed match's time is needed, and other details like the field to play in are left out.

Meanwhile other students are also proposing football matches. The following rule is used to detect when there are enough suitable proposals and schedule the corresponding match:

```
define      FootballMatch(When, Player1, Player2, Player3, Player4)
from        FootballMatchProposal() as FMP1 and
            1-first FootballMatchProposal(Abs(When-FMP1.When)<30min) as FMP2 within 5hour
            from FMP1 and
            2-first FootballMatchProposal(Abs(When-FMP1.When)<30min) as FMP3 within 5hour
            from FMP1 and
            3-first FootballMatchProposal(Abs(When-FMP1.When)<30min) as FMP4 within 5hour
            from FMP1
where       When=(FMP1.When + FMP2.When + FMP3.When + FMP4.When)/4,
            Player1=FMP1.Name, Player2=FMP2.Name, Player3=FMP3.Name, Player4=FMP4.Name
consuming   FMP1, FMP2, FMP3, FMP4
```

Abs computes the modulus of its argument. To simplify, each team has just 2 players. When four proposals with close starting times are collected, a match is scheduled.

Students subscribe for matches in their preferred time window with:

```
Subscribe(FootballMatch, When>[time window start] and When<[time window end])
```

so that they will be notified when a match with start time inside the given window is scheduled. Of course, it is in the interest of a proposing student to subscribe before making the proposal; anyway, a student could subscribe even when he is not going to propose, just to be able to receive notifications about ongoing matches and possibly decide to join one of them at the last moment.

Clearly, this kind of reasoning about proposals and agreements on football matches could be done on many other subjects.

3.2.3 Gatherings of people

Sometimes it could be useful to detect the presence of groups of students stopped and gathered together in an area of the campus. There are many reasons that could cause those gatherings. Happy ones, like for example a concert going on, or a good snack kiosk attracting hungry students, or a stand promoting a new product and giving free gadgets, or the AVIS[7] mobile operations room and so on, could be interesting to students themselves. Other more critical reasons, like an incident just happened, or a fight going on, or a manifestation of any type, could be monitored by some security system.

To know accurately the location of the student the GPS of its device is used. The client application periodically communicate it (eg. every 5 minutes or more) sending to the server an event like:

```
Position(Id=[device's identifier], Latitude, Longitude)
```

where the phone *Id* would probably be its IMEI, and geographical coordinates are expressed as latitude and longitude decimal degrees. To realize whether a student has stopped somewhere, its current position is compared with the previous one; if the two

[7] *Associazione Volontari Italiani Sangue*, the major Italian blood donation organisation

positions are very close the student can be considered stopped and a StoppedPosition event is created:

```
define    StoppedPosition(Id, Latitude, Longitude)
from      Position() as P2 and
          last Position(Id==P2.Id) as P1 within 5min from P2 and
          Abs(P1.Latitude-P2.Latitude)<0.00009 and
          Abs(P1.Longitude-P2.Longitude)<0.00013
where     Id=P2.Id, Latitude=P2.Latitude, Longitude=P2.Longitude
```

The latitude/longitude degree deltas used in this example to compare consecutive positions correspond to linear length of about 10 meters [8].

In alternative, if the GPS of the mobile directly gives the speed, or the client application indirectly computes it, speed can be added to Position:

```
Position(Id, Latitude, Longitude, Speed)
```

with speed given in meters per second. Now the rule that understands if the student is stopped needs only the last detected position:

```
define    StoppedPosition(Id, Latitude, Longitude)
from      Position(Speed<0.1) as P
where     Id=P.Id, Latitude=P.Latitude, Longitude=P.Longitude
```

Whatever is the way StoppedPosition events are created, with the following rule the system can realize, and inform subscribers of, whether significant people gatherings are going on in a particular area of the campus:

```
define    PeopleGathering(CenterLat, CenterLong, DeltaLat, DeltaLong)
from      StoppedPosition() as P and
          30 < Count(StoppedPosition(
               Abs(Latitude-P.Latitude)<0.00027 and
               Abs(Longitude-P.Longitude)<0.00038)
          ) within 5min from P
where     CenterLat=P.Latitude, CenterLong=P.Longitude, DeltaLat=0.00027*2,
          DeltaLong=0.00038*2
```

where *CenterLat* and *CenterLong* are the coordinates of the rectangular area's center, while *DeltaLat* and *DeltaLong* are respectively its height and width[9]. In this example latitude and longitude deltas define a square with semi-side of about 30 meters centered on the last student whose (stopped) position was detected. The threshold to distinguish between numerically relevant and irrelevant gatherings have been arbitrarily set to 30 students.

[8] Considering latitudes of about +45° as in Milan; convertion from latitude/longitude degrees to linear length was done with this tool: `http://www.csgnetwork.com/degreelenllavcalc.html`

[9] Note that the geometry of the area is a *rectangle* when considering its 2-D Mercator's map projection, while it is a spherical *quadrangle* on Earth's surface when considering the 3-D globe. A spherical quadrangle is a region on Earth's surface formed by the intersection of a spherical *zone* (bounded by min/max latitudes) with a spherical *lune* (bounded by min/max longitudes). You can find an example of quadrangle here: `http://www.mathworks.com/help/toolbox/map/quadarea.gif` .

3.2.3.1 Implementation

A demo of this scenario has been implemented in `SOEP`. The `TESLA` rules for *StoppedPosition* and *PeopleGathering* are deployed on `SOEP-Server`. Students send to the server *Position* events, subscribe to *Gathering* events inside the university campus, and are notified when such gatherings are detected by the event engine.

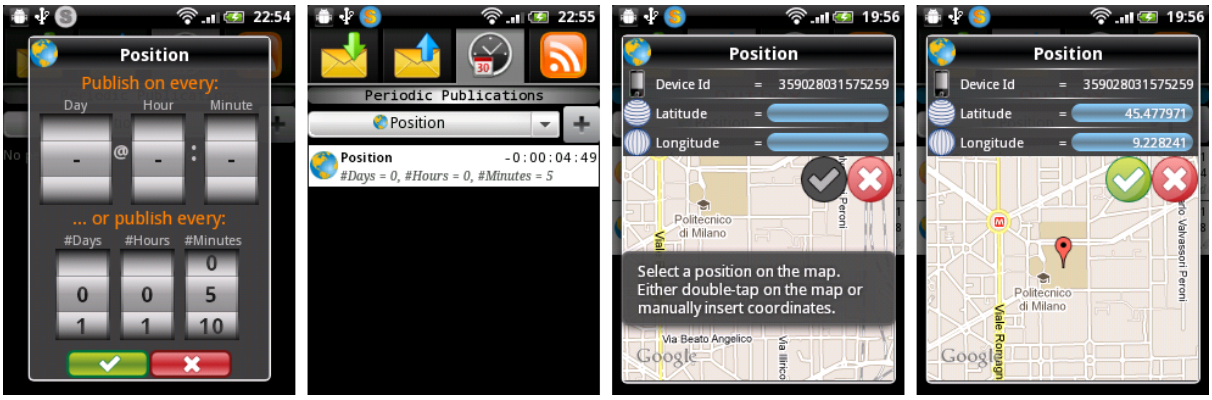

Figure 3.2: Position events in SOEP-Client. From left to right: 1) scheduling of position's automatic publication every five minutes; 2) the periodic publication just scheduled is listed 3) an interactive publication of the position is being started 4) the position to be published has been defined

There are two ways in `SOEP-Client` to publish an event (in this case, a Position): the first one is to schedule the event to be automatically published on a periodic basis (see in the first two screenshots of figure 3.2 the scheduling of the automatic publication every five minutes of the current position); in the second one the user interactively defines a position and sends it (third and fourth screenshot in figure 3.2).

The user can subscribe to gatherings inside a specific geographical area by pinching on the touchscreen a map selection (see the first and second screenshot of figure 3.3 for the subscription to gatherings inside the Politecnico di Milano's campus). When the event engine on the server detects new gatherings from the received positions, a Gathering event is sent to each subscribing `SOEP-Client` (see the last two shots in figure 3.3), which properly notifies and displays it.

3.2.4 Queues at the student office

A student of Politecnico di Milano is never glad of going to the student office: he often has to face endless queues, and the dedicated informative system inside the building is not so efficient. Students are given a paper ticket with the type of the queue, their ticket number, and the current queue length; they then must wait in a place in line of sight with one of the electronic boards, to costantly check for their number approaching. No

Figure 3.3: Gathering events in SOEP-Client. From left to right: 1) a subscription to Gathering events inside *Politecnico di Milano*, *Leonardo* campus is being started 2) the selection area of the subscription has been refined to fit the campus 3) a notification of a gathering inside the campus has been received 4) the received gathering is listed in the inbox

statistics are given before joining the queue, neither after, and leaving the building for a while could mean to lose the turn.

Of course, an informative system cannot do much to shorten service time or improve quality of service on human side, that is, on the work of its employees; nonetheless, it should do its best to inform students about the current traffic situation, helping in self-organization so that queues are faced more consciously.

To reach these goals, a HOEP system that communicate with the students' devices and integrates with existing informative system would be very useful. Students could take their ticket directly with the mobile, events about the queues would be easily notified by the system to the students, or viceversa. Students could wait their turn anywhere because they would check the queue on their phone; short absences from the secretary would be no longer a problem. Everything but the final service of the employee would be in student's hand. In the following, the events to handle this scenario in `SOEP` are shown.

3.2.4.1 Turn Management

When the student joins a queue with its mobile phone, the system assigns him a ticket and broadcasts the corresponding event:

> TicketTaken(QueueID, QueueNum, Time)

where *QueueID* is the identifier of the joined queue, *QueueNum* is the number given to the student in the queue, *Time* is the timestamp of the event.

Similarly, each time a student completes its turn he (or the employee) notifies the system about the completion with:

```
TurnDone(QueueID, QueueNum, Time)
```

On the other hand, when the student for any reason must give up and exits the queue before being serviced, he sends a cancellation event:

```
TurnCancelled(QueueID, QueueNum)
```

TurnCancelled could also be sent automatically by the mobile phone's application when its position has become far enough from the student office building (of course, the student would be warned before).

When a turn is completed, an event is created to inform students about the next number being served (which is the first subsequent one that has not been cancelled):

```
define    NumberServed(QueueID, QueueNum)
from      TurnDone() as TD and
          first TicketTaken(QueueID==TD.QueueID and QueueNum>TD.QueueNum) as TT within
          2hour from TD and
          not TurnCancelled(QueueID==TT.QueueID and QueueNum==TT.QueueNum) between TT
          and TD
where     QueueID=TT.QueueID, QueueNum=TT.QueueNum
```

Every student can see on its mobile phone what number the queue is at, in particular the student whose turn has just arrived is promptly alerted by the application.

3.2.4.2 Waiting Time

To compute the estimated waiting time for a student which is going to enter the queue, both the current queue length and the average service time for each turn are needed.

With the following rule the current size of each queue is updated whenever a new number is served:

```
define    QueueLength(QueueID, Length)
from      NumberServed() as NS and
          last TicketTaken(QueueID==NS.QueueID) as TT within 2hour from NS
where     QueueID=NS.QueueID,
          Length= TT.QueueNum – NS.QueueNum – Count(
            TurnCancelled(QueueID==NS.QueueID and QueueNum>NS.QueueNum)
            ) within 2hour from NS
```

The queue length is obtained as the last assigned ticket number, minus the ticket number of the student currently being served, minus the number of cancellations in between.

To compute the average service time the timestamps in TicketTaken and TurnDone are used. The average is done incrementally as turns are completed, each time averaging the previous estimated time with the new information. After n service time measurements, the system has computed the mean service time $\bar{s}_n$ as:

$$\bar{s}_n = \frac{s_1 + s_2 + ... + s_n}{n}$$

where $i = 1, \ldots, n$ refers to the queue number and each s_i is defined as $s_i = t_{turn\_i\_done} - t_{ticket\_i\_taken}$. Now suppose that a new s_{n+1} is available: the new average service time $\bar{s}_{n+1}$ is computed as

$$\bar{s}_{n+1} = \frac{\bar{s}_n \cdot n + s_{n+1}}{n+1}$$

To do this algebraic operation the number of averaged elements n must be kept, and incremented each time a turn is completed for the considered queue. The *ServiceTime* rule is thus defined incrementally, split in two halves. The first half is used when there are no previous ServiceTime events (that is, at the opening of the student office), and begins the iteration from 1:

```
define    ServiceTime(QueueID, Value, Iteration)
from      TurnDone() as TD and
          last TicketTaken(QueueID==TD.QueueID and QueueNum==TD.QueueNum) as TT within
          2hour from TD and
          not ServiceTime(QueueID==TD.QueueID) within 2hour from TD
where     QueueID= TD.QueueID,
          Value= TD.Time – TT.Time,
          Iteration= 1
```

Instead, the second half is used to incrementally update the previous estimated service time with the information on last completed turn:

```
define    ServiceTime(QueueID, Value, Iteration)
from      TurnDone() as TD and
          last TicketTaken(QueueID==TD.QueueID and QueueNum==TD.QueueNum) as TT within
          2hour from TD and
          last ServiceTime(QueueID==TD.QueueID) as ST within 2hour from TD
where     QueueID= TD.QueueID,
          Value= ((ST.Value * ST.Iteration) + (TD.Time – TT.Time)) / (ST.Iteration+1),
          Iteration= ST.Iteration +1
```

Now that QueueLength and ServiceTime are defined, the computation of the expected overall waiting time for students who are going to join a queue is computed simply as their product:

```
define    WaitingTime(QueueID, Value)
from      QueueLength() as QL and
          last ServiceTime(QueueID==QL.QueueID) as ST within 1hour from QL
where     QueueID= QL.QueueID,
          Value= QL.Length * ST.Value
```

3.2.5 Surveys and Statistics

Surveys are a way to collect judgments directly from people, and they are doubly useful: one time because they help the surveyor (or the commissioner of the survey) to improve things, and another time because they make people feel involved, stimulating collaborative and sympathetic behaviours. This also applies to students. Surveys in Politecnico

di Milano are mostly done in paper form, at the end of each course and at the beginning or in the middle of a lesson, and they include course-specific questions, as well as more general ones. Usually there is not enough time to sufficiently think about every answer, and answers are often rushed and incoherent.

SOEP could manage this. Surveys on mobile phone should be more effective, because:

- the student can decide to postpone them in its break time, and answer quietly
- questions can be much more in number, because more distributed in time
- the tecnological channel is more appealing than writing on paper
- even students could propose surveys and see the results

3.2.5.1 Survey Framework

A survey is started with the creation (done by the system or by a student) of a Survey event:

> Survey(Id, Property, Category, Subject, Question, Answer1, ... , Answer5)

where the attributes are:

- *Id*: a unique identification number for the survey
- *Property*: the property to be analyzed (for example, "TeachingQuality", "Speed", "Cost", "Availability", "Cleanliness", "Kindness", "Openness" ...)
- *Category*: a category for which the property has to be analyzed (for example, "Course", "Teacher", "ClassRoom", "StudyRoom", "Building", "Canteen", "Bar", "Toilets", "StudentOffice", "TeacherOffice", "PrintingFacilities")
- *Subject*: the category's element the survey is dedicated to (for example, the code of a course, the surname of a teacher, the number of a classroom, ...)
- *Question*: the full question (about a specific aspect of the Property for the Subject in the Category) given to students (for example, with Property="TeachingQuality", Category="Teacher", Subject="Fornaciari" a possible Question could be: "Do you think that Fornaciari's lessons are clear enough?")
- *Answer1, ... , Answer5*: five possible answers the student can choose among (usually they would be: Answer1="Very Positive", Answer2="Positive", Answer3="Negative", Answer4="Very Negative", Answer5="Don't know")

When the student's mobile phone receives the survey, it prompts him the question so that he can choose one of the five answers and optionally insert some textual comment, thus creating the answer event:

> SurveyAnswer(Id, IsAns1, ... , IsAns5, Comment)

where:

- *Id*: the identificator of the survey
- *IsAnsX*: equal to 1 if X is the choosen answer, 0 otherwise (of course, only one among IsAns1, ... , IsAns5 is equal to 1)
- *Comment*: optional textual comment

To aggregate results of a survey, a SurveyResult event is defined. The definition is incremental with regards to each new SurveyAnswer received. The first half of the definition is used when a result is created from scratch, that is, no results for that survey has already been fired:

```
define   SurveyResult(Id, Property, Category, Subject, NumAns1, ... , NumAns5, Comments)
from     SurveyAnswer() as A and
         not SurveyResult(Id==A.Id) within 1week from A and
         last Survey(Id==A.Id) as S within 1week from A
where    Id= S.Id, Property=S.Property, Category=S.Category, Subject=S.Subject,
         NumAns1= A.IsAns1, ... , NumAns5= A.IsAns5,
         Comments= A.Comment
```

The second half of the definition contains the incremental mechanism. When a new SurveyAnswer is received, and a previous SurveyResult is found, a new SurveyResult is created with the data of the previous one updated with the last answer:

```
define   SurveyResult(Id, Property, Category, Subject, NumAns1, ... , NumAns5, Comments)
from     SurveyAnswer() as A and
         last SurveyResult(Id==A.Id) as R within 1week from A
where    Id= R.Id, Property=R.Property, Category=R.Category, Subject=R.Subject,
         NumAns1= A.IsAns1 + R.NumAns1, ... , NumAns5= A.IsAns5 + R.NumAns5,
         Comments= R.Comments + A.Comment
```

Given the definition of *IsAnsX* (=1 if chosen answer is X, =0 otherwise), the update rule for *NumAnsX* is a plain sum of A.IsAnsX with R.NumAnsX for every X from 1 to 5. Similarly, the new *Comments* string is the concatenation of the old one with last comment.

3.2.5.2 Survey Analysis

Now that we defined a way to express survey results, some analysis on them can be made. For example, we can make an assessment on the overall teaching quality of courses, based on the judgements of the students. Supposing that surveys on the quality of each course were done (translated in our syntax, they are surveys with Property="TeachingQuality", Category="Course", Subject=[name of a specific course], Question=[a question about the teaching quality of that course]), and that the standard semantics for answers was used (Answer1="Very Positive", Answer2="Positive", Answer3="Negative", Answer4="Very Negative", Answer5="Don't know"), we can establish the overall teaching quality of the courses in Politecnico di Milano:

```
define      TeachingQuality(Score, MaxPossible, PercentageRating)
from        Timer(D==Monday) as T
where       Score=
              +2*Sum(SurveyResult(Property=="TeachingQuality" and Category=="Course").NumAns1)
              +1*Sum(SurveyResult(Property=="TeachingQuality" and Category=="Course").NumAns2)
              −1*Sum(SurveyResult(Property=="TeachingQuality" and Category=="Course").NumAns3)
              −2*Sum(SurveyResult(Property=="TeachingQuality" and Category=="Course").NumAns4)
            within 1week from T,
            MaxPossible= 2*(
              Sum(SurveyResult(Property=="TeachingQuality" and Category=="Course").NumAns1) +
              Sum(SurveyResult(Property=="TeachingQuality" and Category=="Course").NumAns2) +
              Sum(SurveyResult(Property=="TeachingQuality" and Category=="Course").NumAns3) +
              Sum(SurveyResult(Property=="TeachingQuality" and Category=="Course").NumAns4))
            within 1week from T,
            PercentageRating= (Score/MaxPossible +1)*50
```

in which we arbitrarily attributed +2 points for each "Very Positive" answer, +1 point for "Positive", −1 for "Negative" and −2 for "Very Negative"; "Don't know" answers are ignored. *Sum* is an aggregate operator that sums the values of the specified attribute in the given set of events. *Score* is the count of total points made according to these multipliers, while *MaxPossible* is the theorical maximum score that would have been reached if each answer had been a "Very Positive". A *PercentageRating* from 0 to 100 is obtained from Score and MaxPossible this way (where s is Score and m is MaxPossible):

$$\begin{array}{ccccc} -m & < & s & < & m \\ -1 & < & \frac{s}{m} & < & 1 \\ -1+1 & < & \frac{s}{m}+1 & < & 1+1 \\ 0 \cdot 50 & < & (\frac{s}{m}+1) \cdot 50 & < & 2 \cdot 50 \\ 0 & < & (\frac{s}{m}+1) \cdot 50 & < & 100 \end{array}$$

When the teaching quality is critically low (eg. less than 30%), an alarm which reports the quality rating along with a list of the worst courses can be fired:

```
define      LowTeachingQuality(Rating, WorstCourses)
from        TeachingQuality(PercentageRating<30) as Q
where       Rating= Q.PercentageRating,
            WorstCourses= Concat(SurveyResult(
              Property=="TeachingQuality" and Category=="Course" and
              30 >
                ((+2*NumAns1 + 1*NumAns2 – 1*NumAns3 – 2*NumAns4) /
                (2*(NumAns1 + NumAns2 + NumAns3 + NumAns4)) +1) *50 )
            .Subject) within 1week from Q
```

where *Concat* is an aggregate operator which concatenates the value of a textual attribute for each event of a given set. To compute the percentage rating of each course, and verify whether it is lower than the threshold, the same method as in *TeachingQuality* is used here.

On the contrary, an event could report when the overall teaching quality of the courses is undergoing a positive trend:

```
define      TeachingQualityPositiveTrend(AverageGrowth, FinalRating)
from        TeachingQuality() as Q4 and
            last TeachingQuality(PercentageRating≤Q4.PercentageRating) as Q3 within 8day from Q4 and
            last TeachingQuality(PercentageRating≤Q3.PercentageRating) as Q2 within 8day from Q3 and
            last TeachingQuality(PercentageRating≤Q2.PercentageRating) as Q1 within 8day from Q2
where       AverageGrowth= (
              (Q2.PercentageRating – Q1.PercentageRating) +
              (Q3.PercentageRating – Q2.PercentageRating) +
              (Q4.PercentageRating – Q3.PercentageRating)) /3,
            FinalRating= Q4.PercentageRating
consuming   Q1, Q2, Q3, Q4
```

where we arbitrarily defined a positive trend one in which there are four consecutive non-decreasing quality reports.

Given the above survey framework many other analysis, like this on the quality of the courses, could be done.

3.2.5.3 Statistics

Students can also be considered as a big group of raw data suppliers, on which some basic statistics can be made. Unlike surveys, which require user interaction, statistics runs on data automatically acquired from students' mobile phone. In the following some examples are given.

Students' Number It could be useful to know the approximate number of students currently in the campus. To this purpose, the *Position* event defined in section 3.2.3, sent periodically by student's devices, is reused. First of all, the following rule could be used to detect the entry of a student in a particular area of the campus:

```
define      EnterArea(DeviceId, Area)
from        Position(
              Abs(Latitude-[area center latitude])<[area semiheight in latitude degrees] and
              Abs(Longitude-[area center longitude])<[area semiwidth in longitude degrees]
            ) as P and
            last Position(DeviceId=P2.DeviceId and
              Abs(Latitude-[area center latitude])≥[area semiheight in latitude degrees] and
              Abs(Longitude-[area center longitude])≥[area semiwidth in longitude degrees]
            ) within 30min from P
where       DeviceId=P.DeviceId, Area=[area name]
```

where a rule for each relevant area should be defined. *Area* could mean a particular zone inside the campus (Area="Cafeteria", Area="Building21", etc.) or the entire campus itself (Area="Campus"). In short, according to this rule a student has just entered the

area if its current position is inside the area and the previous one was outside. Suppose the opposite rule ExitArea(DeviceId, Area) is defined, too.

Besides the entry to and exit from an area, also the permance on it could be detected:

```
define    InArea(DeviceId, Area)
from      Position(
            Abs(Latitude-[area center latitude])<[area semiheight in latitude degrees] and
            Abs(Longitude-[area center longitude])<[area semiwidth in longitude degrees]
          ) as P
where     DeviceId=P.DeviceId, Area=[area name]
```

Note that while EnterArea and ExitArea are created once each time a student enters or exits a particular area, InArea is periodically created as long as he stays in the area, with the frequency of creation being the same as Position events (eg. 5 minutes).

The most straightforward way to periodically count the number of students inside the campus is through InArea events and the aggregate operator *Count*:

```
define    StudentsNumber(Number)
from      Timer(M%5==0)
where     Number=Count(InArea(Area=="Campus")) within 5min from Timer
```

In alternative, the students' number could be progressively computed by incrementing or decrementing the number when a student enters or exits the campus, respectively. The increment is done with this rule:

```
define    StudentsNumber(Number)
from      EnterArea(Area="Campus") as EA and
          last StudentsNumber() as SN within 12hour from EA
where     Number=SN.Number + 1
```

while the decrement is done with this other rule:

```
define    StudentsNumber(Number)
from      ExitArea(Area="Campus") as EA and
          last StudentsNumber() as SN within 12hour from EA
where     Number=SN.Number − 1
```

Both rules would also need their relative initialization part for when no StudentsNumber has still been detected, which has been left out for brevity.

Telephony A bunch of other statistics can be done on telephony data. Suppose that each student's mobile phone in the campus periodically (eg. every 5 minutes or more) sends information about its telephony services:

```
TelephonyServices(DeviceId, IsCalling, IsDataConnected, NetworkOperator, SimOperator, SimCountry,
NetworkType)
```

whose attributes are:

▷ *DeviceId*: a unique identification number (for example, the IMEI) of the mobile device

- ▷ *IsCalling*: "True" if a phone call was going on during detection, "False" otherwise
- ▷ *IsDataConnected*: "True" if a data connection (that is, an Internet connection) is active on the device, "False" otherwise
- ▷ *NetworkOperator*: name of the currently registered telephony operator
- ▷ *SimOperator*: name of the operator which provided the SIM
- ▷ *SimCountry*: country code of the SimOperator
- ▷ *NetworkType*: the radio technology currently in use on the device (eg. GPRS, EDGE, UMTS, HSDPA, etc.)

A rule can be used to make statistics on students that are currently making phone calls:

```
define    CallingStudents(Number, Percentage)
from      Timer(M%5==0) as T and
          last StudentsNumber() as SN within 5min from T
where     Number=Count(TelephonyServices(IsCalling=="True")) within 5min from T,
          Percentage=(Number / SN.Number) * 100
```

which periodically reports the number of students in the campus who are making calls, both as an absolute value and in percentage.

Other statistics can be done on network operators, for example to know what is the most used one:

```
define    MostUsedOperator(Operator, UsersNumber, Percentage)
from      Timer(M%5==0) as T and
          last StudentsNumber() as SN within 5min from T
where     Operator=MostFrequent(TelephonyServices().NetworkOperator) within 5min from T,
          UsersNumber=Count(TelephonyServices(NetworkOperator==Operator)) within 5min from T,
          Percentage=(UsersNumber / SN.Number) * 100
```

where *MostFrequent* is an aggregate operator which returns the most frequent value for an attribute among a set of events. The rule reports the name of the competition-winning operator, along with the number of users in the campus using that operator and their percentage on total students.

If the most used network operator has very high percentage of users (eg. 90% or more), an interesting monopoly is going on:

```
define    TelephonyMonopoly(Operator)
from      MostUsedOperator(Percentage>90)
where     Operator=MostUsedOperator.Operator
```

As a final example, from the *SIMCountry* statistics on foreign students can be done:

```
define    ForeignStudents(Number, Percentage)
from      Timer(M%5==0) as T and
          last StudentsNumber() as SN within 5min from T
where     Number=Count(TelephonyServices(SimCountry ≠ "IT")) within 5min from T,
          Percentage=(Number / SN.Number) * 100
```

which reports the approximate number of foreign students currently in the campus, along with their percentage on total students.

3.2.6 Administrative Announcements

SOEP could also be used to distribute university's administrative announcements to students. Currently in *Politecnico di Milano* this is done through a dedicated section in *WebPoliself*, a web tool used by each student to remotely manage his career, and sometimes by emails sent to the institutional address. Actually, the announcements done this way are quite static and generic. SOEP could add some personalization in the distribution of announcements - exploiting the potentialities of the smartphones - thus making them more appealing and useful.

A first kind of announcement could be delivered when a student enters or exits a particular area of the campus, or the campus itself. For example:

```
define    EnterAreaAnnouncement(TargetDeviceId, Message)
from      Timer(M%5==0) as T and
          each EnterArea(Area=="Campus") as EA within 5min from T
where     TargetDeviceId=EA.DeviceId,
          Message="Welcome back to Politecnico. The news and events for today are..."
```

for when the student gets into the campus and

```
define    ExitAreaAnnouncement(TargetDeviceId, Message)
from      Timer(M%5==0) as T and
          each ExitArea(Area=="Campus") as EA within 5min from T
where     TargetDeviceId=EA.DeviceId,
          Message="Goodbye and see you soon in Politecnico. We remind you that tomorrow..."
```

for when he gets out of it. The *EnterArea*, *ExitArea* and *InArea* events have been defined in section 3.2.5.3.

A second possible type of announcement is triggered when a particular time is reached, and is delivered to all the students currently inside a particular area. For example:

```
define    TimeAnnouncement(TargetDeviceId, Message)
from      Timer(H==19 and M==30) as T and
          each InArea(Area=="Campus") within 5min from T
where     TargetDeviceId=InArea.DeviceId and
          Message="Politecnico is closing. Hurry up to leave the classrooms!"
```

The area considered in these rules is variable, from the campus itself as in previous examples up to a single building or even room inside the campus. This way, customized messages are possible.

To simplify student's subscription, a single *Announcement* that aggregates all previous kinds could be defined, in three parts:

```
define    Announcement(TargetDeviceId, Message)
from      EnterAreaAnnouncement() as A
where     TargetDeviceId=A.TargetDeviceId and Message=A.Message
```

and

```
define    Announcement(TargetDeviceId, Message)
from      ExitAreaAnnouncement() as A
where     TargetDeviceId=A.TargetDeviceId and Message=A.Message
```

and

```
define    Announcement(TargetDeviceId, Message)
from      TimeAnnouncement() as A
where     TargetDeviceId=A.TargetDeviceId and Message=A.Message
```

so that the student can make a single subscription with:

```
Subscribe(Announcement(TargetDeviceId=[student's device id]))
```

Several more kinds of announcements that exploit other potentialities of the student's mobile phone could be conceived.

4 SOEP-Server

`SOEP-Server` is the server part of `SOEP`. It runs the event engine `T-Rex` on which `TESLA` rules are deployed, it accepts and mantains connections with `SOEP-Client`s and let them publish and subscribe to events; when new complex events are created, as a consequence of a client's publication, they are reported to every subscriber.

In section 4.1 a description of server's features is given; section 4.2 details its architecture and its main execution flows, finally section 4.3 adds some notes on the implementation and on the libraries used to develop it.

4.1 Features

Here is a list of the features of the current version of `SOEP-Server`:

Connections

- ▷ accept new connections from clients (asynchronous)
- ▷ receive data from each connection (asynchronous)
- ▷ mantains alive each connection with heartbeat
  - ◇ periodically send pings (asynchronous)
  - ◇ periodically check heartbeat timeout's expiration from last reception (asynchronous)
  - ◇ close connection when an error occurred or timeout expired

Event Rules

- ▷ deploy `TESLA` rules on `T-Rex` at startup

Subscriptions

- ▷ accept subscriptions
- ▷ accept unsubscriptions
- ▷ remove subscriptions when client disconnects

Publications

- ▷ receive publication of events from clients
- ▷ let `T-Rex` generate complex events according to deployed `TESLA` rules
- ▷ publish generated complex events to subscribers

User Interface

- ▷ simple command-line interface
- ▷ Log file written to disk

`SOEP-Server` was designed with a simple centralized architecture and a TCP connection is opened and kept alive with every client.

4.1.1 Scalability

Despite the centralized architecture, particular attention was paid on scalability and the server should prove stable enough to manage a few thousands of clients, as a university campus like Politecnico di Milano requires on average[1]. On one side, this number of simultaneous connections should be treatable even by a single machine. On the other side, the main factor contributing to scalability is the usage of asynchronous input/output operations to handle each connection, combined with a thread-pool to execute the handlers associated with them.

Indeed, the traditional *thread-per-connection* approach, which requires the creation of a new thread for every accepted connection, would pose a big bottleneck on server's scalability: threads are usually considered heavyweight because their creation, switch and synchronization require lots of computational resources. With this approach, synchronous (that is, blocking) I/O primitives are used, and the only way to obtain concurrency is to spawn new threads. Concurrency is thus tightly coupled with threading: the more concurrency is needed, the more threads the application must spawn. It is easy to imagine how this can rapidly reach the threshold on system resources' usage.

On the contrary, the *asynchronous I/O* model, which requires the help of the Operating System to schedule asynchronous operations, is way more efficient: only a few threads (or a single one, at the most) are needed to alternately start asynchronous operations and execute the associated handler when the OS notifies about the operation's completion. For example, a thread can asynchronously accept a connection on a socket and in the meantime execute other jobs: it is not necessary to block until a client will connect, the OS will notify the thread when this will happen.

4.2 Design

The architecture of `SOEP-Server` is shown in figure 4.1. The main components are `T-Rex` and `Connection`: the former is the CEP event engine, which accepts as input new events received from clients and processes them to create new complex events as output, according to deployed `TESLA` rules; the latter represents an open session with a single client, including all the `Subscriptions` made by the client. This session is not stateful over network connection's drop (a design choice to lessen the management's burden on the server): this means that when a client purposely disconnects or when the connection is abruptly closed due to any error, subscriptions are freed from memory. It is up to the client to send its subscriptions every time it connects to the server.

Client and server exchange data on the network in form of `Packet`s: there is a packet type for every relevant action on events, including `PubPacket` for publications and `SubPacket`

[1] According to the Statistical Office of the Italian *Ministry of Education, Universities and Research* (`http://statistica.miur.it`), in the academic year 2008/09 the students registered at Politecnico di Milano were about 36000, of whom about 21000 in *Leonardo* campus.

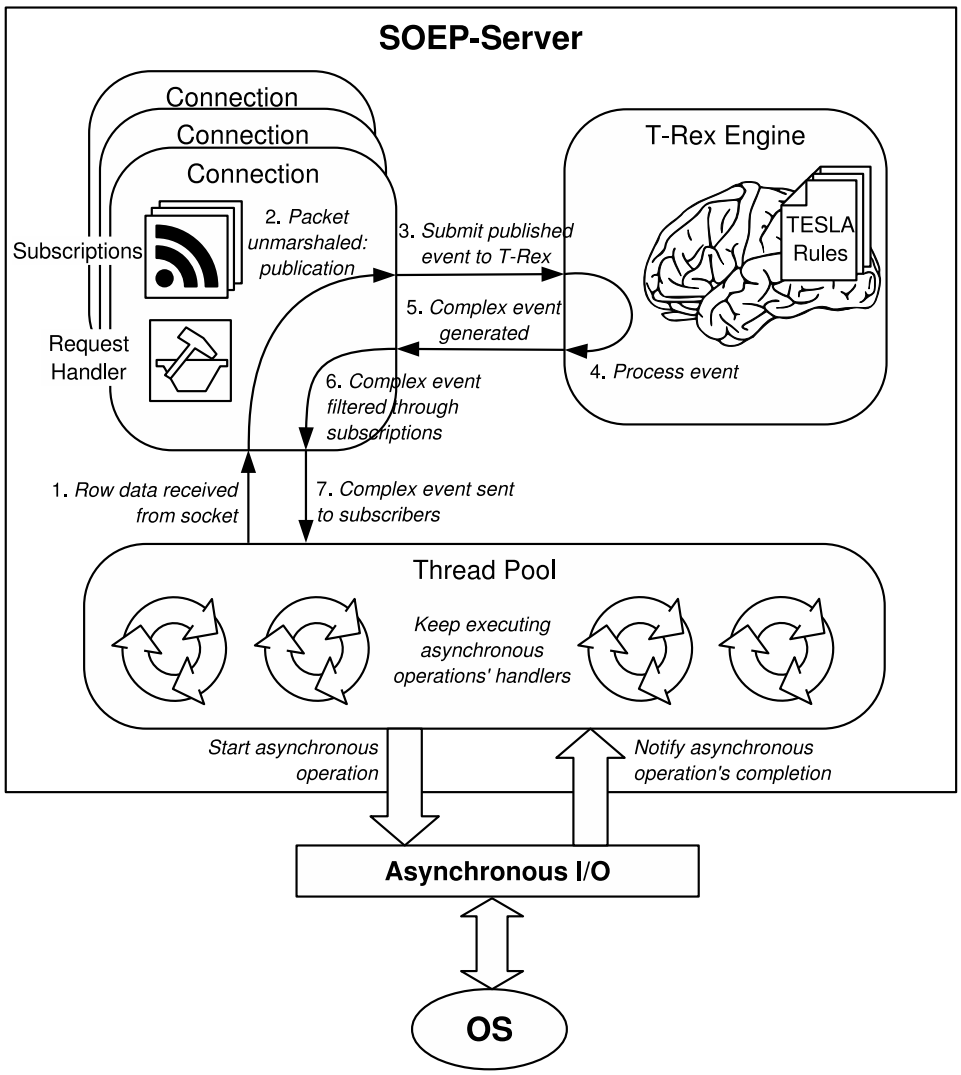

Figure 4.1: SOEP-Server's architecture

for subscriptions. The marshalling algorithms used to flatten packets to bytes to be put on the network, as well as those for the opposite procedure, are obviously shared between client and server. A Connection uses a `RequestHandler` to handle each new piece of data received from the socket: this involves unmarshalling the packet and reacting according to the packet's type. If the received packet is a SubPacket, the corresponding subscription is added to the Connection (if it is not already registered), while if it is a PubPacket the publication must be processed by T-Rex (see figure 4.1), which could generate back a new complex event: in that case the relevant subscribers are first identified by matching the event with all the registered subscriptions, then the event is sent them.

The operations described above are performed by a limited number of threads in a *Thread Pool*, which cyclically schedule asynchronous operations on socket (accept new connec-

tions, receive data, send data) to an *Asynchronous I/O* layer which stands in-between the server and the OS. When any of those asynchronous operations is completed, one of the threads in the threadpool is arbitrarily chosen to execute its associated handler; by starting again another asynchronous operation from the handler, the continuity of the server's work with each connection is guaranteed.

4.2.1 Server's Workflow

The asynchronous flow of execution is unfortunately harder to understand (and to develop on) than the standard synchronous mechanisms, due to the separation in time and space between operation initiation and completion. To clear it up a little more, a flowchart of the main parts of `SOEP-Server`'s workflow is available in figure 4.2. The diagram shows the typical flow of execution of the main components of the server and their interrelations. Dotted arrows correspond to the start of an asynchronous operation, which will eventually (on a later time) lead to the execution of the associated handler, represented with a parallelogram.

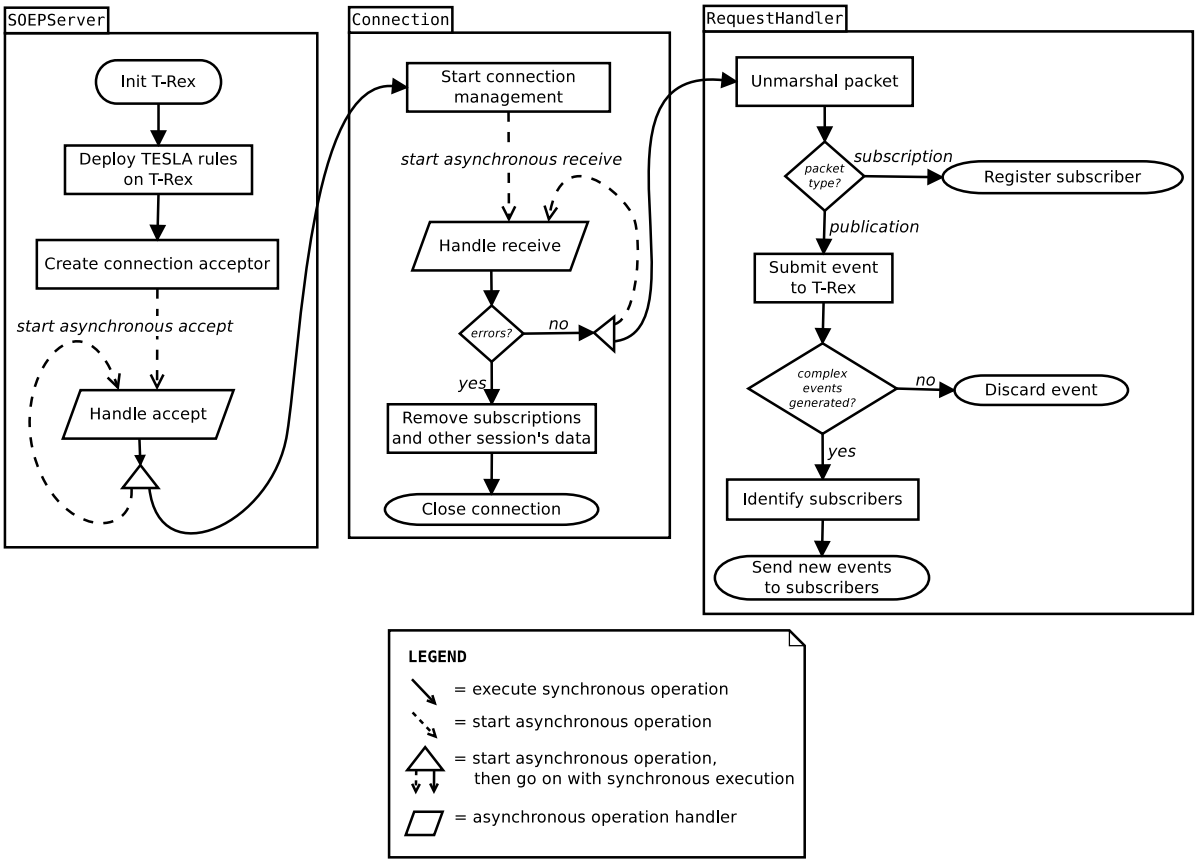

Figure 4.2: SOEP-Server's workflow

At startup, `SOEPServer` initializes `T-Rex` and deploys all the `TESLA` rules on it. After that, it starts the asynchronous *accept loop*: whenever a new client connects, a new

Connection is created to manage the session with that client. The created Connection is assigned the socket to communicate through, and immediately starts the asynchronous *receive loop*: new Packet*s* are iteratively received and processed by Connection's RequestHandler, until a network error is detected. A mention should also be made on the existence (not shown in the diagram because conceptually less important) of two other asynchronous loops: the *ping-receive loop*, which keeps checking the heartbeat timeout's expiration from last reception, and the *ping-send loop*, which periodically sends pings to clients.

4.3 Implementation

`SOEP-Server` is *free* software, released with GPLv3 licence and implemented in C++. It was developed with Eclipse CDT and tested on a Linux machine, but its code is cross-platform. Source code is versioned with SVN and hosted on a repository at Politecnico di Milano.

`SOEP-Server` links with some libraries: the *C++ Standard Library* (including the *Standard Template Library*, STL), *T-Rex* and various *Boost* libraries. As the Standard Library is well known, and T-Rex has been already described in section 2.2.2, section 4.3.1 will focus only on the latters.

4.3.1 Boost Libraries

The *Boost C++ Libraries* (`www.boost.org`) are a large collection of *free* libraries that extend the functionality of C++, most of which are released with the permissive *Boost Software Licence*, which allows them to be used both in free and commercial projects. The collection contains more than a hundred libraries aimed at a wide range of application domains, from general-purpose libraries, to very specific ones, up to libraries primarily aimed at other library developers. They are maintained by an active community of developers, which submit each new proposed library to a collective review process; several Boost libraries have been accepted for incorporation into both the C++ *Technical Report 1* and the upcoming new standard *C++0x*. A peculiarity of Boost libraries is that most of them are header based, consisting of inline functions and templates, and as such do not need to be built in advance of their usage. [48][49]

`SOEP-Server` was tested with Boost version 1.45.0. More specifically, the used Boost libraries were:

Boost.Asio The most important one for `SOEP-Server`, it is a cross-platform library for both synchronous and asynchronous network and low-level I/O programming. Referring to figure 4.1 on server's architecture, Asio provides the *Asynchronous I/O* layer (implemented by the library's `io_service` object). Some of the features of Asio are the management of sockets (TCP, UDP, ICMP and basic SSL),

timers, buffers, serial ports and file descriptors. Another advantage of Asio is a simplification on threads' synchronization: the `strand` facility wraps asynchronous completion handlers so that they are executed serially; this is used in `SOEP-Server` to serialize the execution of each Connection's asynchronous completion handler (while there is no need for synchronization among different Connections). The main disadvantage of the asynchronous mechanism in Asio is an increase in program complexity/readability and in memory usage, because a separate buffer is required for each concurrent operation and its space is reserved for the complete duration of the operation.

Boost.Thread It enables the use of multiple threads of execution in portable high-level C++ code. It provides classes and functions to manage the threads, synchronize them, share data between them or conversely keep separate copies of local data specific to individual threads. It is used in conjuction with Boost.Asio to handle `SOEP-Server`'s threadpool.

Boost.Array STL compliant wrapper for arrays of constant size. Using this class, an array can be created that exhibits the same properties as a traditional array in C++ and in addition conforms to the requirements for C++ containers, so that it can be used with the framework provided by the Standard Library for processing algorithms on containers. It is used in `SOEP-Server` mostly as a fixed-size buffer of bytes (actually, `char`s).

Boost.SmartPointers Smart pointers are objects which store pointers to dynamically allocated (on the heap) objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. Conceptually, smart pointers are seen as owning the object pointed to, and thus responsible for deletion of the object when it is no longer needed. The library provides six smart pointer class templates; the most used in `SOEP-Server` is `shared_ptr`, whose semantics is the sharing of an object's ownership among multiple owners, with the object being guaranteed to be deleted when the last `shared_ptr` pointing to it is destroyed or reset. With proper use of smart pointers, explicit `delete` statements should no longer be necessary.

Boost.Variant The `boost::variant` class template is a type-safe, generic, stack-based discriminated union container, which allows the manipulation of an object from a heterogeneous set of types in a uniform manner. Whereas standard containers such as `std::vector` may be thought of as "*multi-value, single type*", variant is "*multi-type, single value*". The library provides compile-time type-safe value visitation according to the *Visitor* design pattern. It is used in `SOEP-Server` to uniformly manage all types of packets (`RulePkt`, `PubPkt`, `SubPkt`, `PingPkt`, etc.) through a single type (`Pkt`).

Boost.Log It provides various logging mechanisms to an application, and has a modular and extensible architecture [50]. Actually, this library is not yet an official part of the Boost libraries, although it has passed the review and is provisionally accepted. It is used in `SOEP-Server` to log events both to console output and on a log file

stored on filesystem.

5 SOEP-Client

`SOEP-Client` is an Android application installed on students' Android-powered mobile phone whose purpose is to provide them with a portal to many useful services related to university life. These services are based on the contributions from all the students, according to the HOEP model: students are both the sources of the information items (simple events) and the recipients of the new knowledge created from them (complex events) or, in other words, the end users of the system. The application acts as a bridge between each student and `SOEP-Server`, where the events are routed to and processed to generate new information, which in turn is redistributed to students. The mobile phone is supposed connected to the Internet, either through a Wi-Fi channel or the cellular network.

`SOEP-Client` allows the management of events through a graphical user interface: users can publish new events (both interactively or by scheduling them to be automatically published on a periodic basis), define subscriptions for particular events and receive new publications of the subscribed events, generated by the server. The events the application can handle are specified as a collection of *event modules*; each module knows the semantics of its event, and how to manage the creation of a new publication or subscription for it. The details on the structure of events (their type and attributes) are assumed to be shared by clients and server a priori. A daemon manages the connection with the server and exposes an interface to other components through which they can send and receive events.

Before getting down to `SOEP-Client`, some preliminary remarks on Android's architecture are given in section 5.1. Then in section 5.2 a list of the features of `SOEP-Client` will be presented. Section 5.3 describes the application's architecture and main design choices. Finally, section 5.4 concludes with some notes on implementation.

5.1 Android

Android[1] has been chosen as the mobile phones' operating system, because it is a promising open-source project which is gaining more and more consensus both in the industry and among end-users, rapidly establishing as one of the most important mobile platforms. It is attractive to users for its good usability (focused on touchscreen smartphones and tablets), the availability of thousands of apps on the Android Market, out-of-the-box

[1] `www.android.com`

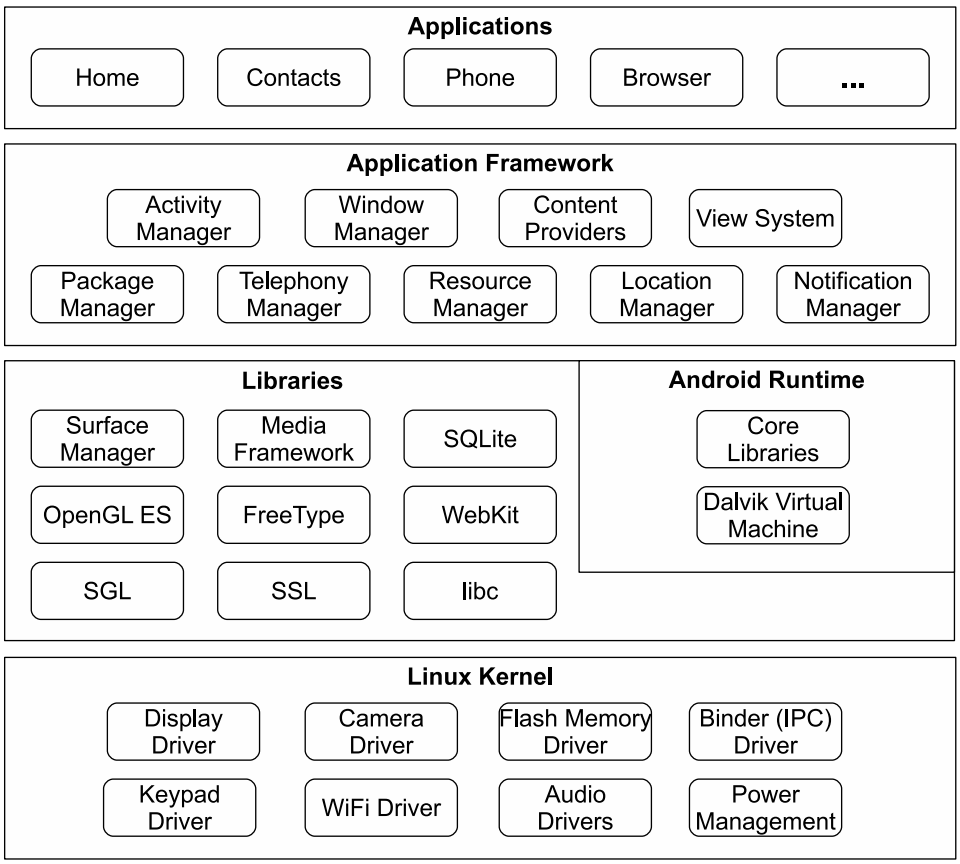

Figure 5.1: Android's architecture

integration with lots of services (especially Google's ones), smart management of hardware resources (battery, GPS, Wi-Fi, 3G, etc.) and last but not least its nice appearance; on the other hand, developers face with a new powerful event-based architecture, tons of good documentation, built-in facilities and frameworks for the most common tasks, together with a development environment and a rich set of tools which significantly ease programming.

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. Its stratified architecture is shown in figure 5.1 [51].

In the following the main levels of the platform's stack are briefly described.

Applications The core applications which are shipped with Android, including an email client, SMS program, calendar, maps, browser, contacts, and others.

Application Framework The high-level component-based Java framework used by the core applications, which is leveraged by developers to build their own application. The architecture is designed to simplify the reuse and replacement of components also among different applications.

Libraries A set of native C/C++ libraries used by various components of the Android system, which are also exposed to developers in a high-level Java-based

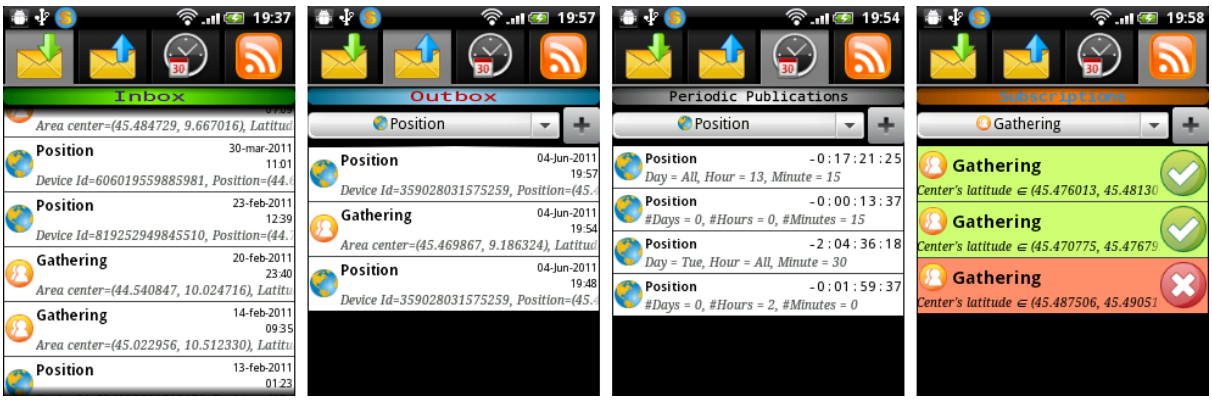

Figure 5.2: SOEP-Client's main views. From left to right: *inbox*, *outbox*, *periodics* and *subscriptions*. Note the application's icon (the circle with a 'S' inside) on the left part of Android's top status bar.

way through the application framework.

Android Runtime Android includes a set of core libraries which implements most of (but not all) the Java Standard Edition's core APIs. It uses the *Dalvik Virtual Machine* [56], which is a particular virtual machine optimized so that every Android application can efficiently run in its own process with its own instance of the VM. Dalvik executes files in the *Dalvik Executable* (`.dex`) format, obtained by converting standard `.class` files, optimized for minimal memory footprint. The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

Linux Kernel Android is built on a Linux kernel and relies on it for core operating system's services such as memory management, process management, hardware drivers, power management, network stack and so on. Drivers for all the hardware equipments embedded on the mobile device are included.

5.2 Features

Here is a comprehensive list of the features of `SOEP-Client`:

Views (see figure 5.2)

- ▷ *inbox*: management of received event publications
  - ◇ list received publications
  - ◇ show/delete/copy-to-clipboard each publication
- ▷ *outbox*: management of sent event publications
  - ◇ list sent publications
  - ◇ show/delete/copy-to-clipboard each publication
  - ◇ publish new events
    - ○ the choice of events depends on the available event modules and their

capabilities
- the UI depends on the module

▷ *periodics*: management of scheduled automatic publications (called *periodics*)
- ◇ list scheduled periodics
- ◇ show/edit/delete/copy-to-clipboard each periodic
- ◇ schedule new periodics
  - the choice of events depends on the available event modules and their capabilities
  - the UI (a nice wheel widget) is the same for all the events
  - periodicity can be *absolute*: defined as the day-hour-minute of the week (and several combinations, eg. every day, every hour, etc.) the publication should be fired on
  - periodicity can be *relative*: defined as an interval expressed in ‹#days, #hours, #minutes›

▷ *subscriptions*: management of event subscriptions
- ◇ list subscriptions
- ◇ show/edit/delete/copy-to-clipboard each subscription
- ◇ temporarily enable/disable each subscription
- ◇ make new subscriptions
  - the choice of events depends on the available event modules and their capabilities
  - the UI depends on the module

▷ *preferences*: management of application's preferences
- ◇ list all options
- ◇ edit each option

Background daemon

▷ manage connection with server
- ◇ connection/disconnection
- ◇ ping and timeout

▷ expose interface to other components
- ◇ to send publications and subscriptions to server
- ◇ to manage periodics
- ◇ to listen for sent/received events
- ◇ it hides low-level networking details

▷ work is batched
- ◇ let the phone sleep most of the time
- ◇ schedule regular wakeups
- ◇ work done on every wakeup

▷ publish periodics
- ◇ check periodics' trigger time on every wakeup
- ◇ publish periodic when its trigger time has elapsed

▷ alert user with sound, light and statusbar notifications
- ◇ application's icon in status bar
- ◇ notification of new received publications

Event Modules

▷ each module manages a particular event
- ◇ supplies information about it
- ◇ shows/creates/edits publications for it
- ◇ shows/creates/edits subscriptions for it

▷ general module framework
- ◇ each module extends the abstract base class `EventModule`
- ◇ `EventModule` provides default implementations for some features, while others must be redefined by subclasses

- ◇ a module is not compelled to provide all the features
- ◇ a list of `Capabilities` is automatically determined for each module to declare what it can do

▷ `EventManager` is used to retrieve modules

- ◇ at startup it loads all the available modules
- ◇ provides several methods to query modules

▷ two demo modules already implemented

- ◇ `PositionModule`: the position of the mobile phone
  - ○ Google Maps to display the position
  - ○ both GPS-based and network-based position detected
- ◇ `GatheringModule`: a gathering of people in a geographical area
  - ○ again Google Maps and position detection
  - ○ *pinch-to-zoom* gesture to select the area on the map

5.3 Design

The architecture of `SOEP-Client` is shown in figure 5.3. The server is placed somewhere on campus infrastructure and opened to Internet (or private intranet) access; clients are Android devices connected to the Internet either through the cellular network or via Wi-Fi infrastructure. A TCP connection is established and kept alive between client and server.

In the following sections the main parts of `SOEP-Client` will be described, with a document structure which actually corresponds to the application's Java packages : the *service* in section 5.3.1, the *activities* in section 5.3.2, finally *event management* in section 5.3.3.

5.3.1 Service

An Android `Service` is a component focused on background work; it does not have a visual interface like an `Activity`, but rather exposes an interface through which activities can control it and post some jobs to be processed on the background.

`SOEPService` is a service used as a bridge between `SOEP-Server` and all the other components of `SOEP-Client`. Its main purpose is to decouple networking from application logic performed by the activities. It autonomously establishes a TCP connection with the server and makes its best to keep it alive, by maintaining a regular heartbeat and recovering it when network failure is detected[2]; Android's `ConnectivityManager` is also leveraged to quickly react to local connectivity's changes, such as the disconnection of a network interface which results in total connectivity loss or in failover to another

[2] As it was explained in section 4.2, this also involves re-establishing again all the client's subscriptions

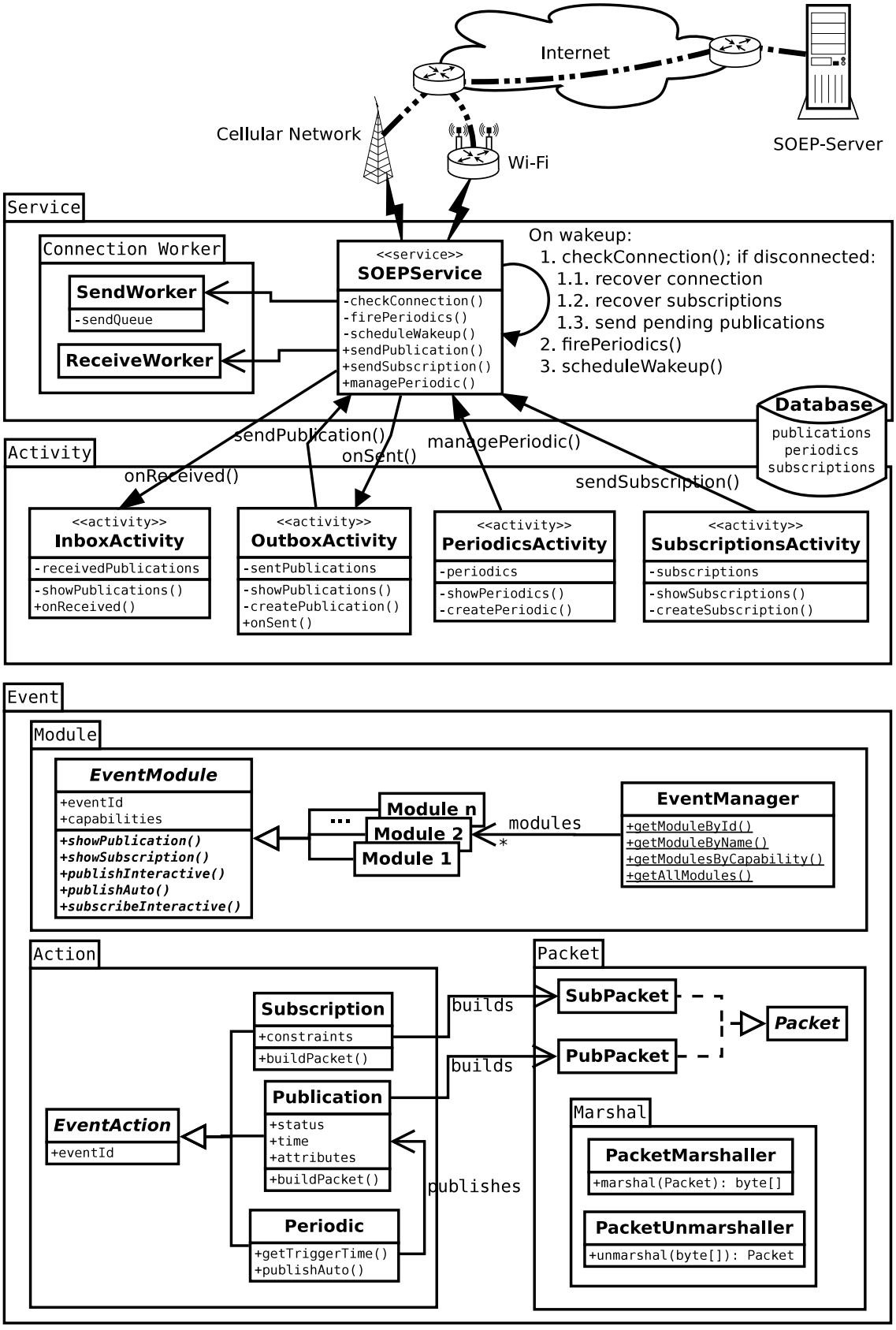

Figure 5.3: SOEP-Client's architecture

interface. The service exposes an API that lets activities publish an event or send a subscription with fire-and-forget semantics: new publications and subscriptions are queued until they can be successfully delivered to server. Activities can also register a listener on the service to be notified of network updates, such as the reception of a new publication, change in connection's status, and so on. Finally, the service takes care of firing *periodics* when their trigger time expires, generating a new automatic publication from them and sending it to the server as for publications interactively created by the user.

To save battery life the service does not run continuously but rather it batches work and schedules regular wakeups through Android's `AlarmManager`; this way, the phone is allowed to sleep most of the time but the OS wakes it up on a regular basis and the service can catch up on work. The work the service iteratively carries out involves:

- checking the status of network connection; the connection could have been dropped either due to an explicit error in trasmission, or because the heartbeat timeout expired; the recovery process requires that:
  - client's subscriptions must be sent again to the server
  - any pending publication stored in the send queue but not yet delivered to the server is sent
- checking periodics' trigger time and publishing the ones whose time has elapsed
- scheduling next wakeup; the wakeup delay is dynamically chosen:
  - a fixed interval is used when the service is connected
  - a binary exponential backoff algorithm with minimum and maximum threshold is used to calculate the delay when the service is not connected; the purpose is to promptly react to sudden disconnection (using a low minimum threshold) but also to avoid premature reconnection requests, which mean battery's waste (using a high maximum threshold)

Once it is started, the service will keep running even when none of the activities are currently in the foreground (because the user switched to another application), until the user explicitly quits the application. A permanent notification of the ongoing service is placed on Android's status bar, while a temporary notification is added whenever a new publication is received.

5.3.2 Activity

An Android `Activity` is the component dedicated to user interaction: it presents a visual interface and reacts to user input. The main activities in `SOEP-Client` are discussed below.

InboxActivity is dedicated to incoming publications. It shows the history of received publications, and gives the possibility to view each one in details. It registers a listener on `SOEPService` to be notified when new events are received and promptly

display them. To display a publication, it queries the corresponding `EventModule` from `EventManager` and delegates to it the task (as only the module knows how to properly display the event's content). Publications (as well as periodics and subscriptions) are fetched from database.

OutboxActivity is dedicated to sent publications. It shows the history of sent publications and lets the user publish a new event. The choice of the publishable events and how to graphically publish them depends on the available event modules. A new publication is sent to server through `SOEPService`.

PeriodicsActivity is dedicated to periodics. It shows the scheduled periodics and the countdown to their next trigger time. User can add a new periodic or edit an existing one. The activity notifies `SOEPService` about the periodics' update: the service is in charge of automatically firing them when necessary. A new periodic is created graphically through a wheel widget which lets the user specify the periodicity of the publication either in *absolute* terms, as the day-hour-minute of the week (and several combinations, eg. every day, every hour; this format has been called *DHM*), or in *relative* terms, as a simple interval expressed in ‹#days, #hours, #minutes›.

SubscriptionsActivity is dedicated to subscriptions. It shows the subscriptions already made and allows the creation of a new one or editing an existing one; it is also possible to temporarily disable a subscription, which will be enabled with a single click when needed. Both the deletion and the temporary disabling of a subscription correspond to an unsubscription, which is sent to server through `SOEPService`; in the second case, moreover, the subscription will be sent unchanged to server when the user enables it again. The choice of the events which a subscription can be made for, and how to graphically create them depends on the event modules installed on the system and their capabilities.

There are other activities, not listed here because of minor importance, as for example `PreferencesActivity`, through which the user can set application's options.

5.3.3 Event Management

The classes of this package are *Plain Old Java Objects* (POJO) that help other components in the management of events. Client and server must agree a priori on the definition of each of those events (its id, name, attributes, and the corresponding semantics); changes in a `TESLA` rule deployed on the server should be followed by a consistent update in the corresponding event module installed on the client.

In the following the main subpackages will be detailed: event *modules* in section 5.3.3.1, event *actions* in section 5.3.3.2, finally event *packets* in section 5.3.3.3.

5.3.3.1 Module

`EventModule` is the core of the module's architecture, subclassed by concrete module implementations. `EventManager` is used to query modules.

EventModule is an abstract class that represents an event module, which must be subclassed by real module implementations. It provides subclasses with a `Capability`-based framework in which they can choose what features to implement or redefine, and consequently the corresponding capability is automatically assigned to the module. A module should be queried about its capabilities before using it. The main features a module can implement are[3]:

- ▷ show a publication (this is the only compulsory feature)
- ▷ show a subscription
- ▷ create or edit a publication interactively (that is, let the user graphically publish an event)
- ▷ create or edit a subscription interactively
- ▷ create a publication automatically (that is, without user's intervention), according to the module's built-in behaviour; this is used with periodics (eg. `PositionModule` scheduled to periodically send phone's position to the server)

Individual modules that perform this kind of management on events are needed because it would be hard to generalize a single abstract model: remember that `SOEP-Client` is meant to be used by students, so each event should create an individual distinguished user experience that conveys the feeling of a portal to many separate services, rather than to plain uniform events' management.

EventManager helps other components to retrieve one or more desidered modules, either querying them by id, or by name, or by specifying the needed capabilities, or directly asking the list of all modules.

5.3.3.2 Action

`EventAction` is the abstraction of an action done on an event: the available actions are `Publication`, `Subscription` and `Periodic`. Actions are widespread throughout the application, manipulated by all the main components (activities, service, modules), and saved on database. They are a high-level concept used internally in `SOEP-Client`, as opposed to `Packet`s which are a low-level representation generated from actions and sent on the network to the server (or conversely, which generate actions when received from server)[4]. In the following the main actions will be briefly described:

[3] Note that these features are relative to the event type the module is assigned to

[4] Note that not all packets are generated from, or generate, actions; for example, a `Periodic` does not directly generate a `PubPacket`, and there is a `PingPacket` used only by `SOEPService` which does not have a corresponding ping action

Publication a publication of an event. It mostly includes the event's type, the event's attributes (key-value pairs), a timestamp and a status (either received, sent-correctly, send-pending or send-failed). `Publication` and `Subscription` extend a particular type of action called `SendableAction`, which represents an action suitable to be sent to the server, by generating a corresponding `Packet`: in case of a `Publication`, a `PubPacket` is generated.

Subscription it represents either a subscription or an unsubscription to an event. It mostly includes the event's type and the event's constraints (attribute-operation-value triples); it also has a flag indicating whether the subscription is currently enabled or disabled. When the subscription is enabled, the corresponding generated packet is a `SubPacket`, while if it is disabled an `UnSubPacket` is built.

Periodic a periodic automatic publication of an event. It includes the type of the event to be automatically published, and a description of the periodicity given either in absolute or relative terms (see `PeriodicsActivity` in section 5.3.2). According to the periodicity set, it computes the next time a `Publication` should be created from it, starting from the last time this was done. `SOEPService` on every wakeup checks what are the periodics whose trigger time elapsed, generates the corresponding publications and send them to the server; the periodic's last trigger time is then updated.

5.3.3.3 Packet

Packets are information units transmitted on the network between clients and server. The most important ones are `PubPacket`, which announces a new publication (either a simple event from a publisher to the server, or a complex event generated from the server and notified to all subscribers) and `SubPacket`, which describes a subscription to an event made by a client[5]. There is also a `PingPacket` used for the heartbeat mechanism which discovers dead connections.

A packet is flattened to an array of bytes to be sent on the network, and the recipient will unflatten those bytes to the original packet. Similarly to `SOEP-Server`, `SOEP-Client` has a `PacketMarshaller` and a `PacketUnmarshaller` which perform the two tasks, respectively. Of course, the marshalling procedure is shared between client and server.

5.4 Implementation

`SOEP-Client` is *free* software, released with GPLv3 licence. As for every Android application[6], its source code is written in Java, while resources are composed by XML files

[5] The opposite packet, `UnSubPacket`, is used for unsubscriptions

[6] Note that an Android application may also have native code written in C/C++ and combined with the other parts in Java through the *Android NDK*, or it may be web-based, that is, composed of web pages either directly opened with the Android browser or integrated in a standard application with the `WebView` widget

and multimedia files (mostly images). It was developed with Eclipse CDT on a Linux machine and tested on a HTC Wildfire A333 with Android 2.2.1, even though the minimum required Android version is 2.0. Source code is versioned with SVN and hosted on a repository at Politecnico di Milano.

In section 5.4.1 a brief outline of the libraries used in `SOEP-Client` is given, while section 5.4.2 details some of the issues found while implementing the application, along with the chosen ways out.

5.4.1 Libraries

Although Android already includes many components and libraries to perform a wide spectrum of common and not-so-common tasks (see section 5.1), there are times when external libraries are needed to fulfill some particular requirements. In this regard, the Android SDK allows the creation of Android *library projects*, standard projects which can be referenced by other application projects to reuse shared code and resources. It is also possible to almost-seamlessly integrate many pure-Java libraries archived in `.jar` files: in this case issues may arise when the library uses a part of the standard Java APIs which are not yet implemented in the core Android libraries. As a last chance, there is a way to integrate native libraries written in C/C++ with Android NDK. [51]

`SOEP-Client` uses a mix of built-in libraries, Android library projects, and pure-Java libraries. In the following these libraries are presented.

SQLite3 a lightweight relational database engine available out-of-the-box in Android. `SOEP-Client` uses it to store on database business data such as publications, subscriptions and periodics. To save user's preferences the `SharedPreferences` framework is used instead, which allows to save and retrieve persistent key-value pairs of primitive data types in a simplified manner.

Google Maps Library an external library which provides access to the Google Maps APIs and enables the integration of Google maps in an Android application. Differently from Android's source code, most of which[7] is open, this library is proprietary and released under the *Google Maps APIs Terms of Service*. Among other things, this requires the developer to register at Google to get a *key* which must be referenced in source code to access maps' services. Most Android-powered devices come with the Google Maps Library installed. The main component of this library is `MapView`, a widget which must be included in a `MapActivity` to display a map with data obtained from the Google Maps service. It captures keypresses and touch gestures to pan and zoom the map, can also be controlled programmatically and overlays can be built on top of it. Moreover, the map can be displayed in a number of modes, such as *satellite*, *traffic*, and *streetview*. [52]

[7] Apart from latest Android version 3.0/3.1 (alias *Honeycomb*), which is currently kept closed by Google and shared only with major manufacturers

`SOEP-Client`'s event modules may use a Google map to display the position of entities referred to by published events, and ease user's manipulation of coordinates or geographical areas.

android-wheel an Android library project developed by Yuri Kanivets, which provides an application with nice scrollable wheel widgets [53]. `SOEP-Client` uses a customized wheel to let the user input the periodicity of a periodic.

TRex-Client-Java this is the Java client adapter of T-Rex (see section 2.2.2). Mostly, it includes Java definitions of the packets used to communicate with the T-Rex engine, and the code to marshal/unmarshal them. `SOEP-Client` extends it with the other packet types used in `SOEP`.

5.4.2 Issues

In this section some of the issues emerged both during the design and implementation of `SOEP-Client` are addressed; a hint about the solution or work-around adopted to tackle them is also provided. Despite Android's architecture being solid and the relative documentation almost always exhaustive, it is still a young platform and rough edges can be found as soon as you venture into more advanced matters. Luckily there is a large community ready to help, whose most notable communication channels are the *Android Developers* mailing list[8] and *Stack Overflow*[9] .

5.4.2.1 SOEPService's startup method

There are two ways to activate an Android's `Service`: it can be *started* or it can be *bound*. A started service usually performs a single operation and does not return a result to the caller. It can run in the background indefinitely, even if the component that started it is destroyed; it is destroyed as soon as any component *stops* it. On the other hand, a bound service offers a client-server interface that allows components to interact with the service, either on the same process or even across processes with interprocess communication (IPC). A bound service runs only as long as another component is bound to it: multiple components can bind to the service at once, but when all of them unbind, the service is destroyed; as a particular case, a component automatically unbinds from a service when it is destroyed. There is a third possibility: a service could be both started and bound; in this case, the service is destroyed only when at least a component has stopped it *and* all the components which bound to it unbind. [51]

As regards `SOEPService`, it was necessary to both start it and bound it, because:

- the activities need the service to expose a rich interface to control it (the service must provide the API described in section 5.3.1), thus they bind to it;

[8] `https://groups.google.com/forum/#!forum/android-developers`
[9] `http://stackoverflow.com/questions/tagged/android`

▷ the service should still go on when the user switches to another application, and should be destroyed only when the user deliberately chooses to *quit* the application. For several reasons (described in section 5.4.2.2), the application's activities can be destroyed when the application is put in background, and consequently their binding with the service is automatically removed: to prevent the service from being destroyed when all the activities unbinds, it should also be started.

In conclusion, all `SOEP-Client`'s activities *binds* to `SOEPService` when they come to the foreground, to obtain access to the service's interface, and at least one of them also *starts* it. When the activities are put in the background to let another application come into play, they *unbind* to the service but none of them also *stops* it. The service will be stopped (and consequently destroyed), only when the user clicks on the "quit" button.

5.4.2.2 Components' memory retainment

Perhaps one of the most striking and difficult to accept change in Android for a novice developer accustomed to standard desktop applications, regards the memory management of the application's components. Indeed, the life cycle of Android's components as *Services*, *Activities*, *Content Providers* and *Broadcast Receivers* is mainly managed by the system: the developer is given a series of *callbacks* to be overridden, which are properly executed by the system on every relevant change of the component's life cycle. Direct consequence of this is that a developer cannot just *close* the application, that is, stop it and free memory: an application can be switched from background to foreground and viceversa, but it is the OS to decide what and when processes will be stopped. This mechanism is used to optimize memory usage and favour processes' recycle. [51]

Another important consequence is that the OS may *kill* our process if memory is required for something else. Of course, there are some guarantees which a developer should carefully exploit to avoid the application's process being unnecessarily killed or at least to take proper countermeasures when it cannot be avoided. The system uses a quite complex priority-based mechanism to choose what are the processes eligible to be killed, which could be simplified in the following four steps (from higher to lower priority)[10] [51]:

1. **foreground process**: a process which contains, among other components, a *foreground activity* (an activity at the top of the screen that the user is currently interacting with); it is considered the most important and will only be killed as a last resort, if it uses more memory than is available on the device
2. **visible process**: a process which contains, among other components, a *visible activity* (an activity that is visible to the user but not in the foreground, such as one sitting behind a foreground dialog) but not a foreground activity; it is considered extremely important and will not be killed unless when it is required to keep another foreground process running

[10] This classification includes only activities and services, which are the most common components; things are slightly different for content providers and broadcast receivers

3. **background process**: a process whose components are neither foreground activities nor visible activities; it may contain *background activities* (activities that are not visible to the user and has been paused) and/or services. A background process is no longer critical, so the system may kill it to reclaim memory for other foreground or visible processes.
4. **empty process**: a process hosting no activities, services or any other components; it will be killed very quickly by the system as memory becomes low

So far only a single process was considered, but an application's components may span over more than one process. For example, a service hosted on a different process may be interacted with through IPC. In case of services there are three more rules to keep in mind for prioritization [51]:

- ▷ if the service is only *started*, its hosting process is normally considered a background process with regards to priority
- ▷ if the service instead (or in addition) is *bound*, its hosting process is never less important than its most important client process. That is, if one of its client processes is a foreground process, or a visible process, the service's hosting process is respectively considered a foreground or visible process, too
- ▷ a started service can be permanently *highered to foreground* priority, if necessary; it will still be theoretically possible for the service to be killed under extreme memory pressure, but in practice this should not be a concern

Going back to `SOEP-Client`, the application is mainly made by activities and services hosted by the same process. To avoid stalling the UI, the service uses a dedicated thread. As the activities and the service share the same process, the process' priority will depend on the activities' status: foreground or visible priority when the user is interacting with them, or background priority when he switches to another application. As for activities, when they are in background they are not serving any purpose and they could be safely killed; actually the main reason to preserve the process from being killed is `SOEPService`, which has a key role in the application. This is almost totally ensured by permanently upgrading it to the status of foreground service. In the remote possibility that Android still needs to kill it, the service is configured to be automatically restarted when more memory will be available.

5.4.2.3 EventModule's activities

As it was explained in section 5.3.3.1, each `EventModule` implementation may provide a way to show, create and edit publications and subscriptions for the event it handles. This means that each module uses, possibly with the help of a common framework, some UI widgetry to interact with the user.

At first, this seemed the ideal context for Android's `Dialog`s: a dialog is a small window that appears in front of the current activity, which gains focus and accepts all user interaction. They are normally used for notifications that should interrupt the user and

to perform short tasks that directly relate to the application in progress. Each module would have used dialogs customized with specific widgets to fulfill its needs. [51]

Unfortunately, the `MapView` widget - which is used to display Google maps in `PositionModule` and `GatheringModule`, as well as presumably in many future modules - cannot be put neither in a standard dialog nor in a standard activity: it needs a specifically designed `MapActivity`, which manages the setup and teardown of the services behind a map (eg. threads which access the network and filesystem in the background to download map's tiles and cache them). Moreover, only one `MapActivity` is supported per process.[52]

As dialogs were thus unusable, the choice fell back on activities. The main issues with activities in this context are:

- differently from dialogs, activities cannot be instantiated from code, instead they have to be declared in the application's XML manifest so that Android can manage their life cycle; but it seemed unfeasible to declare tens of activities in the manifest (one for each module)
- they are heavyweight objects, thus it would be inefficient to have one of them for each module

The adopted solution was to use just two activities: `EventModuleActivity` for normal usage and `EventModuleMapActivity` - which extends `MapActivity` - to be used with maps; one of the two is automatically chosen depending on whether the module needs a map or not. A module does not directly implement the activity (as obviously it is not possible to provide different implementations for the same activity), instead it provides it[11] with a `ModuleActivityJob` which contains all the information about what it should execute, and when. By using only a pair of activities, Android's work on recycling components is favoured.

5.4.2.4 Capabilities discovery

As described in section 5.3.3.1, an `EventModule`'s subclass exposes a set of `Capabilities` according to the features it provides by overriding the appropriate methods of the parent. When a module's method is executed but it does not have the relative capability, a `CapabilityException` is thrown.

The most straightforward way to specify capabilities would be to just let each module declare them at compile time. Anyway, developing the demo modules `PositionModule` and `GatheringModule` with this modality proved to be very error-prone because a developer could (mis)declare a capability, but then forget to override the corresponding method; or conversely, a method could be overridden but the corresponding capability not declared. It probably would have needed a runtime check on capabilities' declaration.

[11] Actually, as there is no easy way to pass a complex object like a `ModuleActivityJob` to the activity, the job is registered at `EventModule` with a unique *key*, and only the key is passed to the activity, so that it will be able to retrieve the job by itself

To simplify things and force correctness on developer's side, a different solution was designed: a `CapabilityRecognizer` which autonomously determines module's capabilities at runtime, by verifying what are the overridden methods through *reflection*. This way, a module does not need to declare capabilities by itself, it just overrides the interested methods. The computational effort of using reflection is limited, because modules are instantiated by `EventManager` at startup once for all (as it is explained in section 5.4.2.5).

5.4.2.5 Module retrieval

To the best of our knowledge, there is not yet a mature plugin framework for Android, something like an integrated OSGi framework which could manage the deployment of both code and resource files locally and remotely[12]. Thus, event modules have been implemented simply as subclasses of the abstract class `EventModule`, and packaged in the codebase along with their resource files. Ideally, even without a plugin framework available, the minimum desirable features would have been:

1. dynamically load local modules in a module-agnostic manner (that is, with as less as possible pre-existing knowledge on the modules to load)
2. dynamically download and deploy new or updated modules from a remote location

Unfortunately the first requirement has been satisfied only partially, and the second one not at all. As regards the first, in a standard Java application it would have been solved by putting all the modules' classes in the same package, then loading at runtime all the `.class` files from the filesystem[13] and instantiating them with *reflection*. However, this is no more possible in Android because it uses the optimized *Dalvik Virtual Machine* [56] instead of the standard one, which converts the `.class` files into a single `.dex` file. As a consequence, at least the modules' names must be known a priori to instantiate them with reflection: indeed, `EventManager` includes a static list of all the modules' classes which is manually hardcoded by developers. At startup, it creates a single static instance of each of them and keeps the references in a special map which indexes modules both by their id and their name. This way, it can quickly serve queries of modules either by id or by name. On the other hand, when it is asked for modules by their `Capability`, it needs to iterate through all the modules to build the list of the ones with the given set of capabilities; anyway, the lists returned are built only once and then cached in another map to be quickly retrieved next time.

As for the second requirement, Android does not provide a facility to easily and securely load code and resources from a remote endpoint. Even if the download of `.jar`/`.apk` files and the runtime load of their content is theoretically possible[14], it would still lack the integration with the other part of the Android framework, eg. for text's localization, display's size management, components' layout inflation, etc. Therefore, no mechanism

[12] See [54] for what so far most resembles - but is quite a long way away from being - a plugin architecture for Android; see [55] for the progress in the integration of OSGi in Android

[13] See this snippet of code as an example: `http://snippets.dzone.com/posts/show/4831`

[14] With a specialized implementation of a class loader called `DexClassLoader`

was designed in SOEP-Client to add modules at runtime. As in general an Android application is of limited size, and in particular in SOEP-Client's scenario the addition of new events is not expected frequently, it is far more pratical to just let Android automatically download from the Market a new available version of the application containing new or updated modules.

5.4.2.6 Pinch-to-zoom gesture

Although there actually exists non-touch netbooks which ship Android[15], according to the standard an Android-compatible device must have a touchscreen [57]; moreover, for many of them the screen is multi-touch. Android provides a framework to manage finger-press events, which has also the built-in capability of recognizing many high-level gestures, such as scroll, fling, double-tap, long-press, and so on. However, it lacks support for the stylish iPhone-derived pinch-to-zoom gesture. As it was needed in GatheringModule to select a geographical area by pinching on the map, a PinchZoomRecognizer has been implemented[16] as an add-on to Android's built-in GestureDetector: it recognizes the pinch by properly tracking the first two fingers' pressure and their movement relative to each other, stopping as soon as a finger is lifted up.

5.4.2.7 Device's standby

To the best of our knowledge, there is no technical documentation on Android's standby management. What it is sure, anyway, is that after some minutes without user input, the device enters into sleep mode and most of the running processes freeze in place, picking up what they left when the device exits from sleep.

Most of Android applications - those entirely focused on immediate interaction with the user - does not need to take into account the standby. On the other hand, applications such as SOEP-Client, which have a service on the background which should be kept running, should take countermeasures. In particular, the freezing of SOEPService would be disruptive because it would stop the connection heartbeat, the reception of new events as well as the automatic publications.

Luckily, Android provides a PowerManager through which wake locks can be acquired on several hardware components (CPU, screen and keyboard), preventing them from falling sleep. SOEPService thus needs to acquire a wakelock every time it is waked up by the AlarmManager (see section 5.3.1), to ensure it can perform all its work before the phone goes back to sleep, and to release the lock when it has done. A WakeLockExecutor was developed to execute jobs serially in a dedicated thread, acquiring a wakelock on CPU when first job is offered and releasing it when last job is completed. A little more difficult was to ensure the execution also of asynchronous jobs: as the automatic publication made

[15] but which are never allowed to have access to the Android Market
[16] With the help of [58]

by a module is asynchronous and is accomplished through a listener, the challenge was to acquire a wakelock until the listener's callback is called. The implemented solution was a `WakeLockDecorator`, a utility class which follows the *decorator pattern* to create a *dynamic proxy* [59] of the given object (in our case, the listener), which acquires a wakelock when it is created and releases it as soon as any of the proxied object's methods (in our case, a listener's callback) is first called.

6 Related Work

As it was described in section 2.1, several different communities brought contributions to IFP, and many use cases were proposed and often implemented. Most of them, however, cannot be considered HOEP systems either because they do not include human operators at all, or because these are not active partecipants in the event processing, and rather have side or supportive roles such as system management or data analysis. Even when there are human producers or consumers, most often their work is individually isolated, that is, the service they are partecipating in brings personal benefits withouth sharing knowledge between partecipants. HOEP, on the contrary, focuses on collective contribution, involvement and cooperation, even though at the end the benefit may be individual. In the following a detailed list of those IFP applications (taken mostly from the CEP domain) is given, and their convergence with HOEP, when there exists, is underlined.

Traditional CEP applications come from the finance's domain. Constant analysis of *stock tickers* [4] is required to identify trends, for example a stock which exhibits a jump in price, or comparisons between different stocks being sold and bought, and so on. In *algorithmic trading* [5][3], automated processes are used not only to monitor stocks, but also to determine when to trade and how to trade: what orders to place, what stocks to sell, etc. *Real-time Profit & Loss* [6][3] shows in real-time how the revenue is transformed into the net income after all the expenses have been accounted for; this is used to track the impact of intraday movements, judge risks and fluctuating exposures in making trading decisions.

CEP has also been used to detect in real-time various kinds of frauds. An example is *cellular phone fraud detection* [7], where a large set of accounts is scanned to examine their calling behavior and to issue an alarm when an account appears to have been defrauded; this can be done eg. by profiling users' behaviour, or looking for known fraud patterns. In *credit card fraud detection* [8], continuous streams of credit card transactions are observed and inspected to prevent frauds. In the context of disaster assistance programs, *Disaster Assistance Claim* [9] aims to identify, in real time before money is dispensed, the processed claims that are fraudulent or unfairly treated and the problematic agents and their accomplices engaged in illegal activities. Event processing may also be used for reasoning about *violations of obligations in contracts* [10].

There are IFP applications in business processes. Event-driven *manufacturing control systems* [13][14] require anomalies to be detected and alerted by looking at the events describing system's behaviour. *Supply Chain Management* [11][3] is aimed at the timely provisioning of goods by reacting to low stock of parts or supplies, ordering, integrating

the logistics process and tracking the status of shipments, warehouse management, etc; for example, RFID-based *inventory management* [12] performs a continuous analysis of RFID readings to track valid paths of shipments and to capture irregularities.

Many use cases were also proposed for transports. As regards air transport, the use of CEP has been indicated for managing the more and more crowded *air traffic*, as a replacement of legacy outdated airspace management systems [15]. RFID-based *baggage-tracking systems* [16][3] improve baggage handling by embedding RFID inlays into the tags attached to checked luggage, and by using the tags to sort and track the bags; tracking events are combined with flight and passengers data to specify and update baggage's routing plans. In rail transport, *freight-cars monitoring* [3] is done with a variety of sensors, from temperature sensors which detect overheating of axle bearings that might lead to derailments, to sensors signaling the position of a given freight car both in absolute terms as well as relative to its neighbouring trains, and sensors monitoring the overall condition of the cargo. Many applications have been found also for road transport, mostly in *road traffic monitoring* [3]. This includes a wide variety of tasks, among which: counting of vehicles during specific time windows for the purpose of *traffic planning*, *access restriction* in limited traffic areas, *recognition of vehicles* wanted by the police, *traffic control* and *road tolling systems*. As an example, *Linear Road* [17] simulates a toll system for the motor vehicle expressways of a large metropolitan area, which uses dynamic factors such as traffic congestion measurements and accident proximity, combined with historical queries, to calculate toll charges. Finally, there are use cases for pedestrian walkways, for example in *foot traffic management* [40]. Road traffic and foot traffic control systems may be considered HOEP systems, see the scenarios proposed in section 3.1.3 and 3.1.4, respectively.

Emergency prevention and response is another possible domain. An example is *fire protection* [18], where fire brigades' resources must be coordinated for effective and quick fire averting, rescuing or protecting of entities (human beings, animals, buildings, etc.) within the fired environment. *Intrusion-detection systems* [19] analyze in real-time network traffic, accesses and data flows in information systems to promptly detect, react and possibly anticipate attacks or malicious behaviour to a corporate network. Similarly, but for physical environments, *automed video surveillance* [20] automatically extracts and notifies about predefined atypical events and behaviours in surveillance videos by means of online video analysis, ontology definition and rule-based engines. In *environmental monitoring* [21], sensors are deployed on field to acquire information about the observed environment, detect anomalies, and predict disasters as soon as possible; for example, an *avalanche warning system* [3] combine information about snow conditions and weather information to detect and predict avalanches. The use of smartphones and other wearable or on-body biomedical wireless sensors is proposed for *personal health monitoring systems* [22], which augment the domain of telemedicine with scenarios such as remote monitoring of the health of chronically-ill or elderly patients at home, emergency response, palliative care. Personal health monitoring systems, even though involve people and make extensive use of mobile phones and other wireless devices, do not totally configure as HOEP systems, because there is no collaboration nor interaction between

patients.

Several systems were proposed regarding information dissemination. *Selective Dissemination of Information* systems [3] were used in the 50's and 60's to allow the distribution of items recently published in abstract journals to be routed to individuals interested in their contents. After the advent of the World Wide Web, they were replaced by modern *Digital Libraries* [23] which serve for the synchronization of library catalogues and use alerting systems that inform the user of the availability of new resources meeting the user's specified keywords and search parameters; alerts can be received through various channels, including email, RSS feeds, voice mail, and instant messaging. Besides academic world, technologies were devised to monitor, collect, filter, aggregate and distribute general web information coming both from *web blogs* [24] and *web feeds* (eg. RSS) [24][4].

Some applications were also proposed in CEP with aspects related to the artificial intelligence field. *Pragmatic Web* [26] extends both the Syntactic Web (describing the form of the information) and the Semantic Web (describing the meaning of the information) to provide methods for users to communicate, agree upon and cooperatively modify ontologies in a practical way. In Pragmatic Web, semi-automated agents form virtual teams or virtual organizations which interchange and reuse knowledge to derive new conclusions and decisions according to changed situations [25]. The *ambient intelligence* paradigm refers to electronic environments that are sensitive and responsive to the presence of people, with small embedded devices which work in concert to seamlessly and pervasively support people in carrying out their everyday life activities. Examples of branches in ambient intelligence are *ambient assisted living* [27], which provide supervision and assistance to elderly people, *smart homes* (eg. [28]), where household devices, appliances, entertainment centers, temperature and lighting control units and home security systems behave intelligently, and *smart cities* [29], which extend the ambient intelligence's concepts to a wider area's infrastructure, supporting users in a variety of situations and contexts, from home, to means of transport, public spaces, work spaces and leisure places.

CEP has also found applications for virtual reality. *Cross-reality environments* [43] serve as a bridge across sensor networks and Web-based virtual worlds, improving people's interactions with each other and with the physical world; in this context, CEP helps to turn raw sensor data generated in the real world into meaningful information suitable for representation in the virtual world. *Personal sensing systems* [41][42], which infer the current user's status (activity, environment and social context) from smartphone's and other PAN devices' readings, can be used to update in real-time the user's avatar on a virtual world. *Multiplayer online games* can benefit from CEP engines, eg. in efficiently distributing the game state between players and minimizing communication with the server [44]. Both multiplayer games and personal sensing systems (especially when combined with social networks, see scenario in section 3.1.10) are examples of HOEP systems.

Finally, there have been some proposals and implementations on context-aware com-

puting. An example is the use of CEP for *museum exhibits* [45], where the visitor's mobile phone interacts with RFID-tagged objects to create an augmented reality environment: the environment around a work of art adapts itself and provide additional contextual information to the visitor, both visual (eg. phone display, media renderers) and auditive (eg. speakers). Mobile *Tourist Information Providers* [46][47] supply their mobile users with feedback about their current location and additional information about related tourist sights, considering time, previous feedbacks, user's interests and travel history. *Position-based social networks* extend traditional social networks with services related to user's position; there are dozens of them available on the market (along with the corresponding mobile application for the main platforms such as Android, iPhone, BlackBerry, Windows Phone 7), but what they all have in common is the possibility for the user to *check-in* to a physical place and share his location with friends, possibly supplied with a textual comment, a photo or an audio/video clip, and see whether there are nearby friends. The simplest example in this category is *Twitter* and its new *Location feature*[1], which allows a user posting a new tweet to include his position; the next is *Google Latitude*[2], which maps user's location on Google Maps and Google Earth, stores the history and make it available through a dashboard (showing trips, frequently visited locations and distance traveled), shares location with Google Talk chat contacts in the status message or publicly on a blog or web site. *Facebook Places*[3] adds the possibility to tag other friends on check-ins, to be alerted when a friend has a check-in on a near location, and includes some commercial features as special offers on near shops, discounts, and so on. Another more evoluted example in this category is *Foursquare*[4], which augments user's involvement with elements of competitiveness: users earn *badges* by checking in at certain locations, with certain tags, for the check-in frequency, or for other patterns such as time of check-in and so on; they are granted a *mayorship* when they check-in to a venue more than anyone else in the recent past; finally, they gain *superuser status* if they are selected for their helpful contributions to the community. Note that also `SOEP-Client` can have similar features (eg. with `PositionModule` and other future modules), but what distinguishes it from cited social networks is the use of a general-purpose rule engine (T-Rex) where events are processed according to ad-hoc rules, rather than ad-hoc engines or no processing at all (resulting in plain publish/subscribe). As each event has its own precise syntax (event id, attributes' name and type, etc.) and semantics (what the event and its attributes represent and how they should be used), each module in `SOEP-Client` have to guide the user to properly instantiate and manipulate the event it is associated with.

The theory of *Human Interaction Management* [33] shares some aspects with HOEP. Anyway, while HIM considers human-driven processes which require innovation and creativity (eg. research, product design, marketing, auditing, merging companies, treating a patient, etc. [34]) and as such cannot be automated with traditional BPM techniques,

[1] See `http://support.twitter.com/articles/78525-about-the-tweet-location-feature`
[2] See `www.google.com/latitude`
[3] See `www.facebook.com/places/`
[4] See `https://foursquare.com/`

HOEP focuses on automated systems whose front-end and back-end are made by people.

7 Future Work

Many improvements in SOEP are possible. So far, SOEP-Client includes two demo event modules (PositionModule and GatheringModule), but many other modules could be added, for example to implement the scenarios proposed in section 3.2. Moreover, to boost its spread among the highest possible number of students, versions of SOEP-Client for the other main mobile platforms (iPhone, BlackBerry, Windows Phone 7) could be released.

As regards SOEP-Server, the reference implementation supplied with this work provides the set of features needed to run and test the system, but it could be extended with other useful facilities. For example, a remote administration tool used by rule managers to connect to the server and deploy rules[1] could be implemented. Moreover, the server has been tested with TESLA rules hardwired in source code, but a parser to parse them from config files stored on filesystem could be added, possibly supplied with a hot-deploy feature to automatically detect the addition/removal of those files and update the corresponding rules at runtime.

SOEP could also deal with *uncertainty*, both at event level and at rule level. About the former, there are situations where event processing may become inexact or inappropriate: there could be uncertainty whether an event actually occurred, or inexact event content (wrong attribute's value, wrong timestamp, etc.), or inexact matching between the event and the situations it tries to describe. This may be due to unreliable, imprecise or even malicious sources, network unreliability which results in events' drop or wrong events' processing order[2], rough data sampling, and so on. As regards uncertainty in rules, the system could be able to distinguish between *deterministic* and *probabilistic* rules: probabilistic rules allow to associate a degree of uncertainty to the complex events created, even in presence of precise input events. This would be highly beneficial to the expressiveness of the language adopted by the application (and consequently by its users).

Another improvement in SOEP would be the introduction of distributed event processing, both on server side and on client side. The server could be distributed on an overlay network made by many processing nodes; rules would be decomposed and each part assigned to different nodes. On the other hand, also clients could cooperate in complex event detection, according to their computational capability: this could probably be applied only to a limited subset of operations (eg. content-based filtering and partial

[1] Note that the server is already capable of receiving new rules in form of RulePackets

[2] Note that uncertainty due to network unreliability in SOEP is strongly mitigated by the adoption of the TCP protocol and an heartbeat mechanism

aggregation of simple events), but the system would get high benefits both in bandwith, as communication with the server would be reduced, and in server's processing burden, as a part of the computation would be offloaded to clients.

`SOEP` will be used as the starting codebase for a part of *GreenMove*[3], a project developed within the Dipartimento di Elettronica e Informazione (DEI) of Politecnico di Milano, sponsored by Regione Lombardia. The project aims to create an electric vehicle sharing service, where electric cars or motorbikes can be booked online, picked up at the specified time and zone, and unlocked with driver's Android smartphone. Every vehicle provides several services to its passengers, and is supplied with an electronic box through which it sends events to the server. Once the vehicle is unlocked, the driver interfaces with it either with its Android smartphone (motorbike) or with an on-board Android tablet (car). The server runs a CEP engine (T-Rex) through which it carries on the vehicles' sharing and monitoring, and related services such as aided navigation and traffic monitoring.

[3]Unfortunately, as the project has just started there is no documentation publicly available yet

8 Conclusion

This work introduced *Human-Oriented Event Processing* (HOEP) as a particular type of CEP in which human operators are active partecipants in the event processing, and whose focus is on human interaction, collaboration and involvement. The *Students-Oriented Event Platform* (SOEP), an example of HOEP system designed and implemented for students roaming in a university campus, was presented and discussed. The server side of SOEP runs a CEP engine on which event rules are deployed to dynamically generate new complex events from those received; a connection with each client is kept, and generated events are routed to subscribers. The client part is an Android mobile application installed on student's phone which can publish, receive and subscribe to events; the events the application can handle are specified as a set of modules: each module knows the semantics of its assigned event and assists the user in creating new publications and subscriptions. To confirm the significance of HOEP, a wide variety of scenarios in HOEP's domain were given, both generic and specific to SOEP.

Although it traces its roots to the old-aged world of events, CEP is a relatively young research branch with a promising way in front of it and many new challenges to be faced: it is our opinion that the approach to human beings and their interactions is one of the keys to interpret such evolution.

Bibliography

[1] O. Etzion, P. Niblett. *Event Processing in Action*. Manning Publications. Aug-2010.

[2] G. Cugola, A. Margara. *Processing flows of information: From data stream to complex event processing*. ACM Computing Surveys 2011.

[3] A. Hinze, K. Sachs, A. Buchmann. *Event-Based Applications and Enabling Technologies*. DEBS 2009.

[4] A. Demers, J. Gehrke, M. Hong, M. Riedewald, W. White. *Towards expressive publish/subscribe systems*. EDBT 2006.

[5] J. Bates. Algorithmic Trading. `http://drdobbs.com/high-performance-computing/197801615` . 09-Mar-2007. Retrieved 14-Jun-2011.

[6] B. Giffords, M. Palmer. StreamBase White Paper. *Real-Time Profit & Loss*. `http://complexevents.com/wp-content/uploads/2008/09/streambase_whitepaper_real_time_pnl.pdf` . Sep-2008. Retrieved 14-Jun-2011.

[7] T. Fawcett, F. Provost. *Activity Monitoring: Noticing interesting changes in behavior*. KDD 1999.

[8] N. P. Schultz-Møller, M. Migliavacca, P. Pietzuch. *Distributed complex event processing with query rewriting*. DEBS 2009.

[9] Kun-Lung Wu, B. Gedik, P. S. Yu, K. W. Hildrum, C. C. Aggarwal, E. Bouillet, Wei Fan, D. A. George, Xiaohui Gu. *Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S*. VLDB 2007.

[10] G. Governatori, Z. Milosevic. *Dealing with contract violations: formalism and domain specific language*. IEEE 2005.

[11] P. Guerrero, K. Sachs, M. Cilia, C. Bornh, A. Buchmann. *Pushing Business Data Processing Towards the Periphery*. IEEE 2007.

[12] F. Wang, P. Liu. *Temporal Management of RFID Data*. VLDB 2005.

[13] E. Y.-T. Lin, Chen Zhou. *Modeling and analysis of message passing in distributed manufacturing systems*. IEEE SMC 1999.

[14] J. Park, S. A. Reveliotis, D. A. Bodner, L. F. Mcginnis. *A distributed, event-driven control architecture for flexibly automated manufacturing systems*. IJCIM 2002.

[15] David Luckham. *The Future Event Driven World: Global Air Traffic Management*. `http://complexevents.com/wp-content/uploads/2007/10/visions-of-the-future-atc.doc` . 18-Nov-2007. Retrieved 24-May-2011.

[16] M. C. O'Connor. *San Francisco Airport OKs RFID Bag-Tracking Pilot*. `http://www.rfidjournal.com/article/view/2629` . 31-Aug-2006. Retrieved 15-Jun-2011.

[17] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker. *Linear Road: A Stream Data Management Benchmark*. VLDB 2004.

[18] J. Pottebaum, A. Artikis, R. Marterer, G. Paliouras, R. Koch. *Event Definition for the Application of Event Processing to Intelligent Resource Management*. ISCRAM 2011.

[19] H. Debar, A. Wespi. *Aggregation and Correlation of Intrusion-Detection Alerts*. In Recent Advances in Intrusion Detection, LNCS 2212. 2001.

[20] T. Geerinck, V. Enescu, I. Ravyse, H. Sahli. *Rule-based Video Interpretation Framework: Application to Automated Surveillance*. ICIG 2009.

[21] K. Broda, K. Clark, R. Miller, A. Russo. *SAGE: A Logical Agent-Based Environment Monitoring and Control System*. AmI 2009.

[22] A. Mouttham, L. Peyton, B. Eze, A. El Saddik. *Event-Driven Data Integration for Personal Health Monitoring*. JETWI 2009.

[23] G. Buchanan, A. Hinze. *A Generic Alerting Service for Digital Libraries*. JCDL 2005.

[24] I. Rose, R. Murty, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Welsh. *Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds*. NSDI 2007.

[25] A. Paschke, H. Boley, A. Kozlenkov, B. Craig. *Rule Responder: RuleML-Based Agents for Distributed Collaboration on the Pragmatic Web*. 2007.

[26] M. Schoop, A. de Moor, J. L.G. Dietz. *The Pragmatic Web: A Manifesto*. ACM 2006.

[27] H. Storf, T. Kleinberger, M. Becker, M. Schmitt, F. Bomarius, S. Prueckner. *An Event-Driven Approach to Activity Recognition in Ambient Assisted Living*. AmI 2009.

[28] Georgia Tech. *Aware Home*. `http://awarehome.imtc.gatech.edu/` . Visited 15-Jun-2011.

[29] A. Buchmann. *Infrastructure for Smart Cities: The Killer Application for Event-based Computing*. Dagstuhl Seminar Proceedings 2007.

[30] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. *Models and issues in data stream systems*. PODS 2002.

[31] D. McCarthy, U. Dayal. *The architecture of an active database management system*. SIGMOD 1989.

[32] R. von Ammon, T. Ertlmaier, O. Etzion, A. Kofman, T. Paulus. *Integrating Complex Events for Collaborating and Dynamically Changing Business Processes*. 2009.

[33] K. Harrison-Broninski. *Human Interaction: The Missing Link in BPM*. Part I and II. `http://www.ebizq.net/topics/biz_opt/features/5779.html` . 2005. Retrieved 25-May-2011.

[34] K. Harrison-Broninski. *A Theoretical Basis for the Management of Human-Driven Processes*. `http://harrison-broninski.com/keith/him/resources/white_papers/A_Theoretical_Basis_for_the_Management_of_Human-Driven_Processes.pdf` . 2005. Retrieved 23-May-2011.

[35] S. Wasserkrug, A. Gal, O. Etzion, Y. Turchin. *Complex event processing over uncertain data*. DEBS 2008.

[36] Jianbing Ma, Weiru Liu, Paul Miller. *Event Modelling and Reasoning with Uncertain Information for Distributed Sensor Networks*. SUM 2010.

[37] G. Cugola, A. Margara. *TESLA: A Formally Defined Event Specification Language*. DEBS 2010.

[38] G. Cugola, A. Margara. *Complex Event Processing with T-REX*. Submitted to JCC. 16-Nov-2010.

[39] G. Cugola, A. Margara. *Low Latency Complex Event Processing on Parallel Hardware*. Submitted to JPDC. 24-Mar-2011.

[40] Hans Gilde. *Event Processing example: Foot Traffic Management*. `http://hansgilde.wordpress.com/2010/01/09/event-processing-example-foot-traffic-management` . 9-Jan-2010. Retrieved 23-May-2011.

[41] M. Musolesi, E. Miluzzo, N. D. Lane, S. B. Eisenman, T. Choudhury, A. T. Campbell. *The Second Life of a Sensor - Integrating Real-world Experience in Virtual Worlds using Mobile Phones*. EmNets 2008.

[42] E. Miluzzo, N. D. Lane, S. B. Eisenman, A. T. Campbell. *CenceMe - Injecting Sensing Presence into Social Networking Applications*. EuroSSC 2007.

[43] N. Dindar, Ç. Balkesen, K. Kromwijk, N. Tatbul. *Event Processing Support for Cross-Reality Environments*. IEEE Pervasive Computing. 2009.

[44] G. G. Koch, M. Adnan Tariq, B. Koldehofe, K. Rothermel. *Event processing for large-scale distributed games*. DEBS 2010.

[45] W. Rudametkin, L. Touseau, M. Perisanidi, A. Gómez, D. Donsez. *NFCMuseum: an Open-Source Middleware for Augmenting Museum Exhibits*. ICPS 2008.

[46] A. Hinze, A. Voisarde. *Location- and Time-Based Information Delivery in Tourism*. SSTD 2003.

[47] A. Hinze, G. Buchanan. *The Challenge of Creating Cooperating Mobile Services: Experiences and Lessons Learned*. ACSC 2006.

[48] Boost.org. *Boost C++ Libraries Documentation*. Version 1.45.0. `www.boost.org/doc/libs/1_45_0/libs/libraries.htm` . Retrieved 18-May-2011.

[49] Boris Schäling. *The Boost C++ Libraries*. Version 1.0 / 01-Apr-2010. http://en.highscore.de/cpp/boost/ . Retrieved 18-May-2011.

[50] Andrey Semashev. *Boost.Log Documentation*. Revised 03-May-2010. http://boost-log.sourceforge.net/libs/log/doc/html/index.html . Retrieved 18-May-2011.

[51] Android Developers. *The Developer's Guide*. http://developer.android.com/guide/index.html . Retrieved 01-Jun-2011.

[52] Google Projects for Android. *Maps External Library*. http://code.google.com/android/add-ons/google-apis/maps-overview.html . Retrieved 01-Jun-2011.

[53] Yuri Kanivets. *The wheel widget for Android*. https://code.google.com/p/android-wheel . Retrieved 01-Jun-2011.

[54] Gabor Paller. *Plugins*. http://mylifewithandroid.blogspot.com/2010/06/plugins.html . Retrieved 02-Jun-2011.

[55] Andy Piper. *OSGi and Android - or how to train your appserver*. www.osgi.org/wiki/uploads/CommunityEvent2010/OSGiCommunity10-Piper.pdf . Nov-2010. Retrieved 03-Jun-2011.

[56] David Ehringer. *The Dalvik Virtual Machine Architecture*. http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf . Mar-2010.

[57] Google. *Android 2.3 Compatibility Definition*. http://source.android.com/compatibility/2.3/android-2.3.3-cdd.pdf . 2010.

[58] Ed Burnette. *How to use Multi-touch in Android 2: Part 6, Implementing the Pinch Zoom Gesture*. http://www.zdnet.com/blog/burnette/how-to-use-multi-touch-in-android-2-part-6-implementing-the-pinch-zoom-gesture/1847 . 16-Mar-2010. Retrieved 01-Apr-2011.

[59] Bob Tarr. *Dynamic Proxies In Java*. http://userpages.umbc.edu/~tarr/dp/lectures/DynProxies-2pp.pdf . Retrieved 27-Mar-2011.